How to Claim a ComputationalFeat

Clémence Chevignard\(^1\), 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 \(C(x) = \text{True}\) 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 \cite{Col86} states that for all integers \(x_0\), the iteration eventually reaches 1: \(\forall x_0 \in \mathbb{N}, \exists n \in \mathbb{N} \text{ such that } \phi^n(x_0) = 1\).

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\( \times \)2\(^{60}\) \cite{Baf21}. 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 fallible, 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, Has01] 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 \( \tau \).

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 \( x \in R \Rightarrow C(x) = \text{True} \). 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\(^\text{6}\) \(\hat{x}\) worthy of reporting using several means:

<table>
<thead>
<tr>
<th>Means</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive reinforcement</td>
<td>Prize (e.g., monetary award) if (\exists \hat{x}) discovered and duly reported by the tester.</td>
</tr>
<tr>
<td>Positive punishment</td>
<td>Penalty (e.g., a fine) if (\exists \hat{x}) unreported by the tester.</td>
</tr>
<tr>
<td>Negative reinforcement</td>
<td>Granting an immunity (e.g., from a lawsuit) if (\exists \hat{x}) discovered and duly reported by the tester.</td>
</tr>
<tr>
<td>Negative punishment</td>
<td>Revocation of a privilege (e.g., sales clearance) if (\exists \hat{x}) unreported by the tester.</td>
</tr>
</tbody>
</table>

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

\section{Preliminaries}

\subsection{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 \(\tau\) that the machine \(M(P, x)\) goes through during computation.

\textit{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. \cite{BCG+13} for the very purpose of proving program execution.

\textsuperscript{6} Formally, a counterexample is \(\hat{x} \in U\) such that \(C(\hat{x}) = \text{False}\).
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 \((P, \text{pc}, \{r_1, \ldots, r_K\}, f, \text{mem}, x)\) where:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
</table>
| \(P\) | the program to be executed  
(considered as a read-only sequence of elementary operations) |
| \(\text{pc}\) | 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 |
| \(\text{mem}\) | 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 \(\text{pc}\), and reads if necessary from the input tape \(x\). A special instruction \textit{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 \(\text{pc}\) 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 \texttt{True} or \texttt{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, \(C_{M_B, T}(x)\)

---

\(^7\) The current specifications can be found here: \url{http://www.scipr-lab.org/doc/TinyRAM-spec-2.000.pdf}. 

5
the wrapped version of the conjecture $C(x)$, is the conjunction of:

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

We implement the wrapping directly in $M_{B,T}$: if an execution exceeds a time limit $T$ or happens to claim at some point more than $B$ memory cells, then $M$ returns $\text{False}$ 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 $H : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$, a machine setup (see Section 2.1) $M_{B,T}$, and constants $\kappa < 2^\lambda$, $0 < \tau < u$. 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

```plaintext
procedure GENERATEPROOF($P, U, \kappa < 2^\lambda$)
2: \hspace{1cm} $R \leftarrow \emptyset$
for all $x \in U$ do
4: \hspace{1cm} Run $M_{B,T}(P, x)$, collecting $\tau$, the sequence of machine states
5: \hspace{1cm} if $M_{B,T}(P, x)$ finishes within the $(B,T)$ resource binding then
6: \hspace{1cm} \hspace{1cm} if $P(x) = \text{False}$ then
7: \hspace{1cm} \hspace{1cm} \hspace{1cm} Report a refutation precious counterexample
8: \hspace{1cm} \hspace{1cm} else if $H([P], x, \tau) < \kappa$ then
9: \hspace{1cm} \hspace{1cm} \hspace{1cm} Append $x$ to $R$
10: return $R$
```

**Verifier.** The verifier runs Algorithm 2 on received data and accepts (returns $\text{True}$) or rejects (returns $\text{False}$). The threshold value $\tau$ 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

\begin{algorithm}
\begin{algorithmic}[1]
\Procedure{CheckProof}{$R \subset U, P, \kappa < 2^\lambda, \tau \in \mathbb{N}$}
\State \If{$R \not\subset U$ or $|R| < \tau$} \State \Return False \EndIf
\ForAll{$x \in R$}
\State Run $M_{B,T}(P, x)$, collecting $\tau$, the sequence of machine states during execution;
\EndFor
\If{$H([P], x, \tau) \geq \kappa$ or $P(x) = \text{False}$ or $M_{B,T}(P, x)$ finishes within the $(B, T)$ ressource binding} \State \Return False \EndIf
\State \Return True
\EndProcedure
\end{algorithmic}
\end{algorithm}

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 $M_{B,T}(P, x)$ is entirely determined by $x$ and $P$, from a computational point of view there is essentially no other way to obtain $M_{B,T}(P, x)$ 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

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

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 $U = \{f(1), f(2), \ldots\}$ 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 $\tau$s. 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?

<table>
<thead>
<tr>
<th>Event</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \notin R$ (failure) returns False</td>
<td>The prover picked an input $x$ outside of the agreed-upon domain $U$. This $x$ can be discarded (removed from $R$) by the verifier; *.</td>
</tr>
<tr>
<td>$H([P], x, \tau) \geq \kappa$ (failure) returns False</td>
<td>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 $x$; *.</td>
</tr>
<tr>
<td>$P(x) = \text{False}$ (failure) returns False</td>
<td>The prover gave a counterexample. At this point we may either accept with a different return code, or reject the entire proof as “$P(x) = \text{True}$ for all $x \in R$” doesn’t hold.</td>
</tr>
<tr>
<td>$</td>
<td>R</td>
</tr>
<tr>
<td>$\mathcal{M}$ aborted (interruption) returns False</td>
<td>$\mathcal{M}$ 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.</td>
</tr>
<tr>
<td>all other cases (success) returns True</td>
<td>This is the normal outcome expected in most runs: $P(x)$ returned True and, in addition, $H([P], x, \tau) &lt; \kappa$. Hence $x$ was included in $R$.</td>
</tr>
</tbody>
</table>

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 \texttt{var=91116}; where \texttt{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 \texttt{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 $\eta$ to generate an $R$ fooling $\text{CHECKPROOF}$ is small, the attacker only needs \textit{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 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 $\tau$ can be replaced by a witness $\omega_x$. Consider for instance a conjecture of the form $\exists \omega_x, C(\omega_x, x) = \text{True}$. Typically, in the case of Collatz, $\omega_x$ 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 $\omega_x$ is the pair of primes whose sum gives $x$. Here, because it may happen that multiple $\omega_x$s can be witness to the same $x$ a common random tape must be used to prevent a dishonest prover trying more and more witnesses 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 $\eta$ (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 $\eta$, 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 $(\omega_x, x)$ 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). Since $q$ increases with $|R|$, there exists a threshold $\mathcal{r}$ such that $|R| \geq \mathcal{r}$ is equivalent to $q \geq q'$. Following standard statistical notation, $q' = 1 - \epsilon$ is the proof’s power, and the proof is more convincing when $\epsilon$ 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 $\kappa/2^\lambda$. The distribution of $N$ is then exactly given by the negative binomial distribution of parameters $p$ and $\mathcal{r}$. This distribution models the number of successes in a sequence of independent and identically distributed Bernoulli trials before a specified number $\mathcal{r}$ of successes occurs. If we observe $\mathcal{r}$ successes when repeatedly performing a Bernoulli trial with success probability $p$, then the probability that there were $n$ trials is

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

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

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

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 $_2F_1$:

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

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

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

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

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

\footnote{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.}

\footnote{See for instance \url{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 \( r \) to be accepted. Because \( q(p, u, r) \) is a smooth function of \( p \) it is easy to set \( r \) to the desired value \( \tau \) 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 \( \eta \)

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 \( r \) simply does not exist for a given \( H \).

Under constant \((u, \tau)\), 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 \( \eta = \Pr[|R| > \tau] \) 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 \( \eta \) as

\[
\eta(p, u, \tau) \approx \frac{1}{2} \erfc\left( \frac{\tau - up}{\sqrt{2up(1-p)}} \right) \tag{2}
\]

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

Remark 12. When \( u \) is large, \( \eta(p, u, \tau) \) has a sharp transition from 1 to 0 when \( p \) is fixed and \( \tau \) 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.
Table 1. \(q, \eta\) for \((p, u, r) = (10^{-3}, 10^{7}, 10_i)\) and \(90 \leq i \leq 104\). Note that \(100i = \frac{i}{100} \times 10^{-3} \times 10^{7}\).

<table>
<thead>
<tr>
<th>(i)</th>
<th>90</th>
<th>91</th>
<th>92</th>
<th>93</th>
<th>94</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q)</td>
<td>(1.188 \times 10^{-24})</td>
<td>(2.894 \times 10^{-20})</td>
<td>(2.374 \times 10^{-16})</td>
<td>(6.656 \times 10^{-13})</td>
<td>(6.4781 \times 10^{-10})</td>
</tr>
<tr>
<td>(\eta)</td>
<td>(\approx 1)</td>
<td>(0.999967)</td>
<td>(0.998657)</td>
<td>(0.977304)</td>
<td>(0.841466)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>(i)</th>
<th>90</th>
<th>91</th>
<th>92</th>
<th>93</th>
<th>94</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q)</td>
<td>(1.243 \times 10^{-24})</td>
<td>(3.003 \times 10^{-20})</td>
<td>(2.444 \times 10^{-16})</td>
<td>(6.807 \times 10^{-13})</td>
<td>(6.587 \times 10^{-10})</td>
</tr>
<tr>
<td>(\eta)</td>
<td>(\approx 1)</td>
<td>(0.999967)</td>
<td>(0.998657)</td>
<td>(0.977304)</td>
<td>(0.841466)</td>
</tr>
</tbody>
</table>

Fig. 1. \(q(10^{-3}, 10^{7}, 10_i)\) (in blue) and \(\eta(10^{-3}, 10^{7}, 10_i)\) (in orange) for \(900 \leq i \leq 1040\).
To understand why, assume that we wish to impose \( \eta > \eta \). For large enough values of \( u \) there always are couples \((p, \tau)\) that make this true; while this problem does not admit an analytical solution, we can solve it numerically, using our approximation: \( \forall \eta \in [0, 1], \exists h_0 \in \mathbb{R} \) such that

\[
\frac{\tau - up}{\sqrt{2up(1 - p)}} < h_0 \Rightarrow \eta > \eta.
\]

For instance, for \( \tau = 0.99 \) we have \( h_0 = -1.64497 \).

We can then express \( p \) as a function of \( \tau \):

\[
p = \frac{h_0^2 + \tau - h_0 \sqrt{h_0^2 + \frac{2\tau(u - \tau)}{u}}}{2h_0^2 + u}.
\]

Decreasing \( p \) below this value will decrease \( \eta \) below \( \eta \). Increasing \( p \) will decrease \( q \); therefore this is the best value of \( p \) we can choose for any given value of \( \tau \), under the constraint on \( \eta \).

Since \( q \) is an increasing function of \( \tau \), and because we do not wish the verifier to test all inputs, the best value of \( q \) is \( \overline{q} \) which is reached for \( \tau = u - 1 \). Therefore, when lower bounding \( \eta \) 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 \( \eta(p, 2u, \tau) \); indeed, \( \Pr[|R_0| + |R_1| \geq \tau] = \eta(p, 2u, \tau) \), being the sum of two independent binomial variables of parameters \((u, p)\).
In other terms, extending the computation by using a second hash function replaces $\eta(p, u, r)$ and $q(p, u, r)$ by respectively $\eta(p, 2u, r)$ and $q(p, 2u, r)$ 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 $R_i$s, 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 $r$ 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 $N > r$ drops to $q(p, u, r)^2 \leq q(p, u, r)$.

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 $\eta$. 

Fig. 2. Graph of $\eta$ as a function of $\eta$, for $u = 10^{10}$ (blue line). The dashed red line represents $1 - \eta$. Plotting for $u = 10^3, \ldots, 10^{20}$ yields an identical graph.
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 \( \approx 500 \) 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: \( u = 10^7 \), \( \eta_0 = 0.99 \), \( p = 10^{-4} \), \( r = 10^3 \), 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 \( \eta_0 \), the prover should claim to have checked \( u \) inputs when in reality it has checked \( (1 + \Delta)u \) inputs where

\[
1 - \Delta \geq \frac{\alpha(1 - p) + \sqrt{(1 - p)(\alpha^2(1 - p) + 2r)}}{pu} + r \quad \text{with} \quad \alpha = \text{erfc}^{-1}(2\eta_0).
\]

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 \( \text{Obfuscate}(i, P) \) 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 \( \rho \):

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

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

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

We hence see that for a given \( u, \psi, \rho, \epsilon \) several trade-offs between the size of the proof \( O(\gamma \pi) \) and the work performed by the parties (resp. \( O(\gamma u) \) and \( O(\gamma \pi) \)) are possible as a function of the parameters \( p, \pi \).

\(^{13}\) With accuracy \( 2^{-100} \).

\(^{14}\) Typically by using \( i \) as a seed to a PRNG.
Algorithm 3 Prover’s Algorithm

```
procedure GenerateParallelProof(P, U, κ < 2λ, (γ, τ) ∈ N^2)
2: R ← ∅
4: I ← ∅
4: i ← 0
while |R| ≠ γ do
6:  P_i ← Obfuscate(i, P)
7:  R_i ← GenerateProof(P_i, U, κ)
8:  if |R_i| ≥ τ then
9:      Append R_i to R
10:  Append i to I
12: i ← i + 1
12: return R, I
```

Algorithm 4 Verifier’s Algorithm

```
procedure CheckParallelProof(R ⊂ U^γ, I ⊂ N^γ, P, U, κ < 2λ, (γ, τ, t) ∈ N^3)
2: if |R| ≠ γ or |I| ≠ γ or max(I) > t then
3:     return False
4: for all j ∈ I do
5:     P_j ← Obfuscate(j, P)
6:     if ¬CheckProof(R_j, P_j, κ, τ) then
7:         return False
8: 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
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 $\kappa$. 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\footnote{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.”} 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
Fig. 4. $q$ as a function of $N$ for $u = 10^6$. Different colors correspond to different values of $\kappa$. 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 $U$s.
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 $M$ 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
$t_{i+1} = t_i + (h \mod 256)$. 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.

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,


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


