Sequential computations verification

Jacek Karwowski[1]

[1]GOLEM

8 January 2018

Abstract

We present a way to verify sequential programs execution in untrusted environment. One of the biggest challenges one faces while creating a distributed, unregulated market for computing power is the problem of verification. In the traditional cloud computing use case, end user does not need to worry about the soundness of the computation - the traditional law, and, additionally, the reputation of the company doing computation at stake, guaranteeing that the result will be honest. However, in the case of open, anonymous market, there is a strong economic incentive for providers of compute to return incomplete or even wrong results, with the benefit of doing more work in a given time. Therefore, one needs a system which will be able to check whenever the answer returned is the correct one.

In the case of easily parallelizable computations, it can be done via computing the small part of the solution and comparing it with the result returned by provider. In the case of small and medium-sized deterministic tasks, it can be done via TrueBit scheme. Here, we tackle the problem of large, sequential, non-deterministic tasks, with particular emphasis on neural networks.

We present an algorithm for verification, under assumptions of "rationality" of providers (i.e. not taking actions with negative expected utility) and regularity of the task (i.e. that the difficulty of calculating each step is independent of the particular value of all previous ones)

Let us stress that the problem here is not the one of verifying that the code is correct (what's done using testing & methods of formal verification), or that the result is correct (because the wrong result can result from a mistake in the code), but that the execution of the program really corresponded to what was in the code.

1 Introduction

Let us imagine Alice - machine learning enthusiast - with a carefully crafted neural network model for weather forecasting. As the weather can be thought of as chaotic system, the model is a sequential one - informally speaking, it's execution cannot be parallelized. As Alice doesn't have access to her own machine at the moment, she wants to train the model in the cloud.

But, there is a problem. Without any means of verifying it, doing computation in cloud requires some trust in the Provider - the entity doing computations. And since the market introduces economic incentive for doing as much work as possible, this creates a great incentive for cheating. If Alice requests 100 epochs of training for his network, and the Provider does only 80 of them, it potentially allows him to do 20% more work, and, at the end, earn more money.

Currently, it is not much of a problem, since the market is dominated by big companies[1]. They value

[1]such as Amazon's AWS or Microsoft Azure

trust of the customers much more than potential benefits of such practice, besides operating under the law, which prohibits such behaviors.

But the situation can change quickly with the development of distributed compute market, such as Golem [2], with anonymous or otherwise untrusted Providers - potentially offering computational power much cheaper than big companies do. In such setting, it is essential to have some means of verifying the integrity of execution.

2 Overview

In the first section, we describe the setting, formalizing such notions as "task" and "sequence of steps transitions".

We then describe the algorithm, using a "black box" notion. "Black box" is essentially an interface, providing means to commit to partial solution. There can be numerous different implementations of this interface - in the main paper we focus on the case when "black box" actually constitutes a connection to the requestor machine, although we also describe implementations based on blockchain usage.

The full algorithm essentially boils down to "calculate a step of computation, commit to this partial solution, receive a decision if this step should be saved for later correctness checking, repeat". By using an economical model of exchange between provider and requestor of the task, we show that just by checking a very small (about 2-10) steps of computation, one can make sure that cheating is not profitable for provider and that the only strategy of positive expected value is the honest one.

We then describe a set of various problems and attacks, together with possible solutions to them. The crucial one is a cyclic buffer attack, which can be described as "calculate a buffer of solutions, cycle through them indefinitely". It is possible, because we only have the notion of locally-correct computation, and in this attack, cheating on one step gains the attacker much more than one step of value. We currently don't have a general solution to that problem, only some heuristics for a special cases of computational tasks (streaming computation and some machine-learning related ones). We also discuss the problem of requestor stealing the solution and disputation in case of a conflict (since verification cannot be done on blockchain, there has to be some external entity to do it off-chain). There is also a problem of hashing - since every step of computation has to be committed to, provider has to hash the whole state of the program very often - and it can be prohibitively expensive, for example if the state is 10s of GBs on the GPU (so it has to be transferred to CPU for hashing).

We conclude the paper with a series of appendices:

- Appendix A describes a way to implement a black box entirely on the provider's computer. It is done by making the decision function (deciding if a particular state should be saved or not) very expensive to calculate. At the cost of 2-fold increase in computational power for a particular task, and also slightly modifying the structure of the task (the length of computation is now only probabilistically equal to the length specified by requestor during task creation) we show that it is in fact possible to have such a setting.
- Appendix B shows how large can steps transitions be, in case of simple neural networks, and how much can they be compressed - experimentally verified answer is that not much.
- Appendix C contains details about the proof-of-concept implementation of the algorithm described in the paper

- Appendix D offers some remarks on the cyclic buffer attack solution. It doesn't contain a definite answer, only some observations we made - it is our hope that it will help someone solve the problem.

3 Related works

The most relevant publication in the area is TrueBit [7]. It is creating a whole framework of secure, verifiable computation. Although the Ethereum [8] already makes it possible, it puts a tight limit on the task complexity - users can only execute very small functions, since every instruction has to be executed on all computers in the network. Authors of TrueBit develop a scheme which makes it possible for potentially arbitrary large computations, by using partly off-chain verification.

However, the algorithm presented there requires few assumptions which make it impossible to use in many of popular use cases. The programs are required:

1. to be deterministic. The scheme allows for randomized algorithms to be used by providing precomputed seeds, but it is not taking into account uncontrolled sources of randomness. It is an important assumption, since it allows the whole Merkele-tree based part of verification game. In our experiments, it proved to be quite a big problem, because of floating point rounding problems (that's why there is a need for a special virtual machine - see the next point)

2. to be implemented in the special TrueBit Virtual Machine. Even if the machine would be essentially identical to some broad used standard (not the case of Google Lanai, but authors mention that in the future it might be for example WebAssembly[7]), if the end user wants to use some external library - it has to be reimplemented to work on the virtual machine.

3. to be limited in size. The scheme writes data to blockchain, and said data has to be much smaller than typical programs states - such as 10s of gigabytes for modern neural networks.

We present here a solution which doesn't require such strong assumptions, making it possible to:

1. compute non-deterministic tasks, including use of multi-threaded and GPU based applications

2. use many off-the-shelf programs and frameworks, providing that they support dividing the computation into many steps and dumping the state to disk after every step

3. increase the size limit of a possible state significantly, limited only by the network throughput

We do not dwell much on the implementation side of Smart Contract part - for example when writing "the dishonest Provider's deposit is burned" we just assume that we have system capable of that.

Since we developed our algorithm independently, we use slightly different nomenclature, but for a reader already familiar with TrueBit, it might be helpful to think of our Providers as Solvers and Requestors as Task Givers.

4 The black box algorithm

4.1 Problem specification

Let us forget for some time about the specifics of the use case described and focus on more formal problem description. We have two agents, Requestor $\mathbb{R}$ and Provider $\mathbb{P}$, the first submitting an algorithm F consisting of N (where $N >> 1$) steps $F = (f_i)_{1<i<N}$, together with some input X. Provider has to calculate

$$Y = (f_n \circ f_{n-1} \circ ... \circ f_1)(X)$$

. We can also assume that the algorithm is not parallelizable[2] - there is no other way to calculate that other than by doing N steps:

$$\begin{aligned} X_1 &= f_1(X_0 = X) \\ X_2 &= f_2(X_1) \\ &... \\ Y = X_N &= f_N(X_{N-1}) \end{aligned} \tag{1}$$

where each X_i is a full state after i-th step of the algorithm[3] and that one step of computation is relatively cheap (eg. $O(1)$ of them can be easily recomputed by $\mathbb{R}$ during verification phrase).

The basic intuition for verifying if $\mathbb{P}$ really done the computation honestly is to check some steps of the computation. If $\mathbb{P}$ saves each of the $X_1, ..X_N$, then it is possible for the $\mathbb{R}$ to check if, for some indices $I \subset \{1, 2, ..N\}$ he really gets

$$f_i(X_{i-1}) == X_i$$

for $i \in I$.

Why is checking only the subset of work effective? We will analyze that later on, for now let's just informally say that the probability of spotting dishonest behavior increases exponentially with the size of the subset I.

However, storing and sending all of the intermediate steps is often not possible. We would like to have some way of selecting only a subset that will be stored and sent to $\mathbb{R}$ after the computation finishes. That's where we need a *black box*.

4.2 Algorithm description

Let's say we have a black box, which has three special properties:

1. Some part of its is opaque (i.e. whoever holds it, cannot see the whole state of the box)
2. When something is put inside, there is no way of getting it out.
3. The box can generate pseudo-random numbers in the range $[0, 1]$, where the generator is based on part of it internal state (possibly that part that is hidden from the user).

With the help of such a device, we can implement an algorithm as follows:

First, $\mathbb{R}$ creates an empty box. Then, he gives it $\mathbb{P}$, together with the task to compute, the one-way cryptographically secure hash function H and difficulty parameter ϵ.

Being at state n, $\mathbb{P}$ runs one step of the task, which gets his from state X_{n-1} to X_n, using data X. he then puts the pair $(H(X_{n-1}), H(X_n))$ into the box. After receiving a random number from the box, he checks if it is smaller than ϵ[4]. If it is, he saves the whole state transition (a pair (X_{n-1}, X_n), together with appropriate hashes, to disk. If it is not, he proceeds with execution.

At the end of the computation, $\mathbb{P}$ gives back both the box and all saved triples of states. $\mathbb{R}$ then can:

1. Check if he got all the pairs of states that had to be saved (ie, generates hashes of $(H(X_i), H(X_{i+1}))$ using the same procedure as the box, and whenever he gets back a number less than ϵ, he checks if he received corresponding state transition from the Provider.
2. Check if the hashes of saved triples of states (X_i, X_{i+1}) match the hashes saved in the box.

[2]If it was parallelizable, then procedure of (probabilistic) verification could go as follows: calculate some small, random part of the solution, then check it agains the output send by the Provider.

[3]We are using here Haskell-ish notation, meaning that the state contains also the state embodies also side effects

[4]It is standard, POW-like scheme. If other words, if the output h of H is a 32-bit number, instead of comparing it with ϵ we can just require that it h starts with some predefined number of zeros. More of schemes like that [here]

3. Check if these state transitions that he got make sense. So, he simply loads the state X_i and checks if the state X'_{i+1} he got of it "makes sense". It doesn't have to be identical to X_{i+1} (since there could potentially problems with floating point operations non-determinism, threads-nondeterminism, various other sources of randomness etc), but since we are talking about very small step, states X_{i+1} and X'_{i+1} ought to be very close to each other - differences won't have time to accumulate. (what does it mean to be "very close" really depends on implementation - for example, for X_i being matrices, it can a condition that L_2 norm on $X'_{i+1} - X_{i+1}$ is smaller than some δ). The exact procedure for verifying this depends strongly on the particular task.

5 Correctness

The scheme described above guarantees with some probability p depending on the ϵ, N and frequency of cheating c (probability that $\mathbb{P}$ will return X_i not similar to "true" X'_{i+1}) that Requestor spots a dishonest Provider.

If the Provider cheats, e.g. computes the transition incorrectly, and puts the hash of it to the box, and gets a response indicating that he has to save the step - he cannot re-compute it now, since he already precommited with the previous hash. Reminder: we assumed there is no way of getting things out of the box, so there is no way to delete the previous hash. Since the decision if the state should be saved or not is based on box internal state, and we assumed that part of it is opaque, it means that from the Provider's perspective, it cannot be predicted, so we can expect that the dishonest transitions and decisions to save state will be independent as random variables.

So, we can calculate the probability

$$\begin{aligned}
&\Pr(\text{dishonest state transition is saved})\\
&= \Pr((\text{Saved states} \cap \text{Dishonest states}) \neq \emptyset)\\
&= 1 - \Pr((\text{Saved states} \cap \text{Dishonest states}) = \emptyset)\\
&= 1 - \Pr(A_{pN} \cap A_{\epsilon N = \emptyset})
\end{aligned} \tag{2}$$

where we can think of A_n as randomly (uniformly) chosen subset of n elements from set of N elements. So, the answer is of course $1 - \binom{N-p\epsilon}{pN} / \binom{N}{p\epsilon}$.

Equation in this form is quite complicated, but we can obtain a simpler formula by approximating it from below by $1 - (pN)^{\epsilon N}$ [5] where ϵN is the number of states we check.

We get that the probability of not being caught decreases exponentially with the number of states checked. As before, now we will also postpone - until the last section - the full analysis of why it is enough, and for now just assume that this our potential good solution.

6 Problems & solutions

As we left some questions behind, not really relevant to the core of the argument, this is the time to answer them.

6.1 How do we get the black box?

It is quite difficult to implement, since if we give it to the Provider, he can just copy it and then replace one copy with some previous checkpoint of it - effectively erasing the content of the box and violating the assumption. he can also check the contents of the box, see the random number generator, run it by herself and know in advance what when he will have to save the state - computing honest step transition only then, he breaks the system.

So, there are a few solutions:

[5] by drawing saved states *with* instead of *without* replacement

1. The black box is an API to a program running on the Requestor computer. All assumptions are then met, but it requires Requestor to be online during whole algorithm execution, it requires constant network communication, slowing down computations, and so on. But, not arguably it is the simplest way to implement.
2. The box is on a blockchain, i.e. "putting something to the box" means adding it to the blockchain. The random number generator can be implemented just as hash function taking next (or next nth) block as seed. In this case, the assumptions are also met, and it doesn't require Requestor being online. The drawbacks are that the adding to blockchain and waiting for the next blocks can be time consuming (but if storing previous k states is possible, then can be done, $\mathbb{P}$ keeping a running tail of computations states).
3. The box is on the boxes server. Potentially, there can be an external Provider, or a whole market of black boxes depots, since hosting them is very cheap (low entry barrier in terms of computing power, so high potential supply) and many users would need it. Although it also solves the problem of having Requestor online,, it still has the same problems with network communication overhead etc, and in addition, we are introducing a notion of a trusted third party to the scheme, which is highly not desirable.
4. The box is on the Providers computer. This one is quite complicated to implement, it has its own problems (e.g. on average it doubles the computation power needed for task), but in principle can be done. More details about that in the appendix A.

6.2 How many states need to be to saved?

How many states should be saved to be quite sure that the whole task is done correctly? In some situations, we can just lower ϵ drastically, making saves of state transitions much more frequent and increasing the chance to spot the dishonesty exponentially, but it will be equal to 1 only if we take $\epsilon = 0$, saving virtually every state transition, defying the whole purpose of external Provider. And this only works in the problems where states are very lightweight, the problem boils down to raw computational power, and the solution is very easy to check.

But, in many situations, we cannot afford saving many states, since in algorithms such as neural networks, the state is very big (eg, tens of GBs, as in the modern networks trained on GPUs [5]). It appears that to solve problem posed in this paragraph, we have to analyze the economics of cheating, which is done in the last section of this paper.

6.3 What about non-deterministic computations?

There can be two sources of non-determinism: intensional and unintentional.

Intensional ones include usage of pseudo-random number generators. In that case, Requestor can just send a whole list of one-time seeds in the input data, effectively removing any randomness from computation.

Unintentional sources of non-determinism include threads usage, floating point numbers rounding, garbage collector behavior, etc. However, these can be possibly managed by using appropriate similarity function, as described before - Requestor will not check if the state he got after transition is identical to one he got from Provider, but whether it is sufficiently close. Since we are splitting computation in small steps, the errors won't have time to accumulate.

If we are going to do GPU computation, there are also problems with GPU nondeterminism. In that case, we can either use special GPU modes, allowing deterministic computations, or stick to measuring if the output state is good enough, as described before.

6.4 What if the dishonesty is spotted?

It depends on the particular system (market). There can be a system of trust, which will decrease, or some financial penalty, such as burning the deposit.

The important part is that every message between Requestor and Provider is signed and recorded by both parties, so that the whole communication (including problem formulation and the solution) can be forwarded and verified by third parties.

As it is quite separate problem, we don't dwell into specifics of how such system would work in here. For more on that, we refer to [**?**].

6.5 How to prevent Requestor from stealing solution?

The scheme presented here has a drawback that Requestor receives part of the result (some state transitions) to verification before the whole transaction is finalized. But, a state transition in the last step is basically a full result. Therefore, there is an opportunity for Requestor to just claim the result was wrong, refuse to pay or otherwise compensate Provider for his work and steal it.

The solution to this problem has to be found outside the framework described in this paper. Either it has to be some escrow service, reputation system, a "court" of some sort, that the Provider has access to and which can somehow resolve the conflict. It would be hard to do as a Smart Contract, since blockchain requires determinism assumption. In this paper, we are only interested on how to get the indisputable evidence that the dishonesty occurred, not how to enforce honesty - and if every bit of communication between Provider and Requestor is signed during task execution, it is quite easy to show if either party has been dishonest.

6.6 Cyclic buffer attack

The solution that we presented is based on checking the solution locally and not globally. It can be exploited in a following way: Provider calculates a buffer of honest solutions of length k, namely $(X_i)_{1<i<k}$ - these are just first k steps calculated honestly.

Then, he simply pretends that this buffer, cycled over, is the full solution, e.g. when asked for step n, he returns $X_{(n \mod k)+1}$.

That kind of attack vector is very dangerous, since it allows attacker to greatly reduce the number of dishonest steps, at the same time also keeping the required amount of honest work low.

In the last section we analyze the cheating economics, and prove that with appropriate reward/penalty system, cheating can be made unprofitable for Provider. However, we use the assumption that cheating in $\frac{1}{k}$ of steps is equivalent to saving oneself $\frac{1}{k}$ work. But - it is not the case here! By cheating only in $\lfloor * \frac{n}{2} \rfloor *$-th step (doing one state transition dishonestly), Provider can save half of the work, by using precomputed buffer of solutions. It is not a problem in cases where the task is primarily memory-intensive, but it is when we are dealing with mostly-computing-intensive tasks. Unfortunately, we did not find a satisfying complete solution to that problem. We have, however, identified two partial solutions, which protect the scheme in special cases of streaming computation, many machine learning algorithms and deterministic computations.

1. Deterministic computations
   The important insight here is that the black box, when deciding if the state transition (X_{n-1}, X_n) should be saved, takes into account only the data constituting this state transition - e.g., the only input to the function deciding if the transition should be saved is the pair (X_{n-1}, X_n).
   What it means is that if the data is the same, the decision will be identical. So, if the Provider feeds the same data (the cyclic buffer) n times, Requestor will get the exactly same states n times, separated by exactly the same number of steps. Then, it is trivial to spot that the cyclic buffer attack was used and take appropriate steps.
   It is very important that the computation has to be deterministic. Since the decision function takes into account only the hash of state transition, it is very easy to fool it - just by changing one bit in the "after"-state. However, with the assumption that computation is deterministic, it cannot be done.

2. Streaming computation
   What we call streaming computation here is a computation that uses streamed data. An example of such system is Streamr [6]. If the function that Provider is computing depends non-trivially on the input data, every state transition is different, and there is no possibility to compute a cyclic buffer.

3. Machine learning algorithms
   In the case of machine learning algorithms, we usually do not have the advantage of having deterministic output (because of nondeterministic GPU behavior [3], multithread paradigm that is often used, etc). Iterative machine learning algorithms, such as gradient descent [4], use the same data in every step of computation, making a perfect setting for a cyclic buffer attack. So, the solution is to break that symmetry. First, let's define a step, as it is used in this paper, as some fixed number of batches feed in to the algorithm (the network). Then, every step has to be different, meaning that we feed in different batches of data. As the data will be different, the state transitions will also differ. From the point of view of implementation, it means that together with initial dataset, Provider also gets the ordering of batches to be used.[6]

We also explored some potential paths to the general solution. The most promising one was to use an idea from deterministic computation, eg that the black box decision to save state is based only on the hash of particular state transition. What we would like to do, is to weaken the condition of the determinism. After all, many programs work *almost*-deterministically, meaning that the there are only small differences in how the program will behave being run several times. We formalize and explain this idea in the Appendix D.

On the final note - in many settings, it would be possible to alleviate the problem of cyclic buffer directly. After all, such a transition, let's say from step number 1000 to step number 1 (and then again, and again, and again...) would really stand up in the log. In other conditions though, detecting cyclic buffer would be very subtle - e.g, in the case of solving differential equation, when solution is not converging, it is possible to see the cyclic states in a perfectly normal, honestly computed setting.

6.7 Memory problems

When we enter the world of neural networks computation, there is no denying that we are dealing with great amounts of data, possibly tens of gigabytes [5]. Each step of the algorithm requires Provider to hash the whole state of the program, which means transferring all that data from GPU to RAM, then calculating the hash, and putting it in the box.

In general, let's say that the state is just a very long list of numbers.

What we could do instead of hashing the whole thing would be to calculate hash of just a small part of the data (which part - it would be decided randomly, based on previous box answer). Unfortunately, to this scheme being secure we would need yet another assumption, namely - that constructing state X_{i-1} from the given state X_i, such that $F_i(X_{i-1}) = X_i$ is hard. This is because a potential attack on that scheme would be to just generate a random state, calculate its hash, and then, when box asks for save, simply calculate the state that could lead to that one.

The problem is, that many computational models used in machine learning don't meet this assumption - for example, it is possible to generate a neural network that, after an update, produce a network that has a particular subset of weights. It is sufficient to generate all the weights randomly, except for one layer, and then solve that layer analytically (i.e. solve a system of linear equations - for example, by inverting the matrix).

The proposed solution is to hash one full pass, but limit oneself to state of just one example from the minibatch - the state consisting of forward pass for this example, vector of gradients for all examples (sum of the respective gradients) in the minibatch for final layer, and backward pass for this example. It would require $2 *$ (size of the network) memory, which for a network like ResNet[5] is about 1GB, but it is better than a full 12GB of memory for the whole network (updates for all examples in the minibatch).

[6]But won't shuffling dataset every epoch break something down? Empirically, it slows down computation indeed, but also helps networks generalize [1]

As a side note, we have also done some experiments on compressing the states. Details are in the Appendix B, but the general conclusion was that the compression on neural networks models doesn't work at all, so there is no possibility to reduce the memory cost of the task that way.

7 Economics of cheating

We will now investigate how does cheating look like from economic point of view - for example, how do the incentives look like, and what rules (such as penalty for cheating) should we introduce to counter bad incentives, such that the rational behavior (e.g. a behavior which maximizes the expected value of gain) will be to be honest at all times.

7.1 Some formalism

We will look at the situation where Provider $\mathbb{P}$ computes some task for Requestor $\mathbb{R}$. After the $\mathbb{P}$ does all the work, he sends the results to $\mathbb{R}$, which then verifies them. If anything is detected to being done wrong (we will say that this is a situation where *Requestor catches cheating Provider*), the penalty is executed on Provider. If nothing is detected to be wrong, the payment to the Provider is made. The work consists of M subtasks (no assumptions about them being independent or not independent, in particular - the whole task can be non-sequential one). Let's say that there is a probability c that the Provider will cheat at some particular subtask, and that the probabilities that he will cheat on different subtasks are independent. The Requestor will verify exactly $s \leq M$ subtasks. Then, the *cost* of the whole task, e.g. the cost induced on the Provider during task (can be though as the electricity cost, the hardware opportunity cost etc) will be K (assuming that it is distributed evenly between subtasks). On the other hand, we have the Provider profit margin R_+. In other words, the total price that Requestor pays for the task is $K + R_+$. In our scheme, we will also have a (monetary) penalty for cheating Provider, which will be denoted as R_-. All costs are fixed numbers. Then, the the probability of Requestor catching cheating Provider is $p = \Pr(s, c, M)$. By single letter E we will denote the expected value of costs for Provider (if negative, it means that the Provider is earing, if positive - he's losing money).

Then, we have

$$E = p * \text{Cost}_{\text{ being caught}} + (1 - p) * \text{Cost}_{\text{ not being caught}}$$

Since Provider does $(1 - c)M$ subtasks honestly and cost of one of them is $\frac{\text{cost of all of them}}{\text{number of them}} = \frac{K}{M}$, then the Provider always pays a cost of $(1 - c)M * \frac{K}{M} = (1 - c)K$.

If Provider is caught, then he pays the cost of all honest steps, plus the penalty cost R_-, so we have

$$\text{Cost}_{\text{ being caught}} = (1 - c)K + R_-$$

If the Provider will not be caught, he pays the cost of all the steps he's done honestly, but gets the payment $K + R_+$ from the Requestor, so we have

$$\text{Cost}_{\text{ not being caught}} = (1 - c)K - (K + R_+)$$

We can now write expected cost as

$$\begin{aligned} E &= p((1 - c)K + R_-) + (1 - p)((1 - c)K - (K + R_+)) \\ &= p(R_- + R_+ + K) - R_+ - cK \end{aligned} \tag{3}$$

Since probabilities of cheating at particular tasks are independent, cM of them are done dishonestly and Provider is caught by Requestor if Requestor finds checks even one dishonest subtask, we can calculate probability of being caught as

$$p = 1 - (1 - c)^s$$

if Requestor samples subtasks to verification with replacement, or

$$p = 1 - \frac{\binom{(1-c)M}{s}}{\binom{M}{s}}$$

if he samples them without replacement (which is the obvious way, from the implementation point of view). However, the case with replacement is a pessimistic estimate of the case without replacement, and for big enough M and small enough c is a good approximation, so for the simplicity we will assume that Requestor samples them with replacement. Substituting p in the equation for E we get

$$E = (1 - (1-c)^s)(R_- + R_+ + K) - R_+ - cK$$

What we want is to check whether we can choose penalty R_- and number of subtask checked s such that the expected value E will be negative only of $c = 0$ (eg, only honest Providers will earn).
As we see, the solution doesn't depend on M at all. It means that the number of subtasks Requestor has to check is constant with respect to the size of the task (the number of subtasks) and only depends on the total cost, probability of cheating, the profit margin and the penalty cost.

If we want cheating to be unprofitable (with all variables except s - number of subtasks checked - fixed), we have to solve the inequality $E > 0$ for s:

$$\begin{aligned} 0 &< E \\ R_+ + cK &< (1-(1-c)^s)(R_- + R_+ + K) \\ s &> \log_{1-c}[-\frac{R_+ + cK}{(R_- + R_+ + K)} + 1] \end{aligned} \tag{4}$$

7.2 Intuition

Let us say again what the result we present in here is about: it shows, that if we have a task consisting of millions of subtasks, we don't have to verify all the parts of it, or even any small "percent" of it - if we fix its economic value and other economic factors, such as the penalty cost, the part that needs verification is constant and potentially very small, potentially about 2. To see the intution that drives that result, let's first note that we can assume it's never beneficial to the Provider to cheat all the time (since if he would do that, and we are guaranteed to check at least one result, he will be caught and lose at least the computing power used for communication, if nothing else). Let's assume that the Provider already calculated first n steps honestly. He then made an expense of $\frac{nK}{M}$. If he calculates the rest dishonestly (potentially not spending anything on it), and - for simplicity let's assume that we, as a Requestor, check only one subtask - he has $\frac{n}{M}$ chance of winning $K\frac{M-n}{M}$ and probability of $\frac{M-n}{M}$ of losing $K\frac{n}{M}$, so his expected reward is 0. If we instead check two subtasks instead of one, it is clear that the probability of detecting it raises (even if slightly), the probability of not detecting it - falls down a bit, and the expected value is now greater than 0 (meaning that this dishonest Provider will lose money). In other words - Provider, computating things honestly, invests in the task, and his risk exactly balances Requestor's risk of not detecting dishonesty - that's where we get this constant number of verifications needed.

7.3 Practical usage

Let's see what happens in couple of special cases:

1. If there is no penalty and no profit margin, then for $s = 1$ we have

$$E = cK - cK = 0$$

. It means that if Requestor only checks one subtask, Provider bears the same cost when cheating and when not.

2. As above, but with $s = 2$, we have

$$E = (1 - (1 - 2c + c^2))K - cK = cK - c^2K = c(1 - c)K$$

It means that when Requestor checks two subtasks, Provider's best strategy is to behave honestly at all times or behave dishonestly all the times.

3. Some example realistic values are

$$\begin{aligned} c &= 0.1 \\ K &= 100 \\ R_+ &= 50 \\ R_- &= 200 \\ s &= 2 \end{aligned} \tag{5}$$

and we have $E = 6.49$, meaning that the rational Provider will not cheat.

4. If both penalty cost and profit margin are large (for example, the same order of magnitude as cost of the task K) and $s = 2$ we have

$$\begin{aligned} E &= (1 - (1 - 2c + c^2))(3K) - K - cK \\ &= 6cK - 2c^2K - K < 0 \end{aligned} \tag{6}$$

It means that the optimal strategy for Provider in this setting involves cheating. The K component dominates, so we would have to make the $(1 - c)^c$ relatively small, so s increases (asymptotically) logarithmically, but with a very large constant.

Obviously, the las situation is bad, as if we find ourselves in it, it is beneficial for Provider to cheat.

Fortunately, there are some reasons to believe that we won't find ourselves in this situation. From this point onwards we will assume that $R_- = K$ for simplicity.

1. It is quite reasonable assumption that starting cheating is not free, but it requires some effort: in other words, there is some minimal value of c_0 and that no Provider will cheat with probability less than c_0.

   If we substitute quite reasonable values

$$\begin{aligned} c = c_0 &= 0.05 \\ K = R_- &= 100 \end{aligned} \tag{7}$$

(which is equivalent to the assumption that nobody will cheat with probability less than 5%) and assume $R_+ = 100$, we get that the minimal value of $s = 9$. If we say that the profit margin is lower, eg $R_+ = 10$, then we get $s = 1$.

2. We can assume that, if the computing market will exist, and it will be free, it will reduce profit margins significantly (since it will be very easy to enter the market). This is backed up by empirical observations - significant profit margin reduction happened in Bitcoin and other cryptocurrencies computing markets [**?**]. If we take the limit in the inequality for s we get

$$\begin{aligned} s &= \lim_{R_+ \to 0} \log_{1-c}[-\frac{R_+ + cK}{(R_- + R_+ + K)} + 1] \\ &= \lim_{R_+ \to 0} \log_{1-c}[-\frac{\frac{R_+}{K} + c}{2 + \frac{R_+}{K}} + 1] \\ &= \log_{1-c}[-\frac{c}{2 + \frac{R_+}{K}} + 1] < \frac{1}{2} \end{aligned} \tag{8}$$

So, we see that in the limit (as market grows to be efficient) Requestor needs to check only $s = 1$ subtasks.

For example, if we have w

$$\begin{aligned} c &= 0.05 \\ K &= 100 \\ R_- &= 100 \\ R_+ &= 10 \end{aligned} \tag{9}$$

then $s = 1$ is enough to make cheating unprofitable.

7.4 Conclusion

One note is that from the formula for

$$E = (1 - (1 - c))^s (R_- + R_+ + K) - R_+ - cK$$

it follows that no matter how big the s will be, there is such $c_0 > 0$ such that setting $c < c_0$ makes cheating profitable (if we use sampling with replacement - otherwise, if we set $s = M$, there is no c such that cheating is profitable).
Finishing this section, let us recall that the whole analysis was done with assumption that the Provider is rational in a very strict sense - eg, he seeks to maximize his (expected) profit from the task.
This assumption can be violated in many ways - one particularly worth attention is when cheating is used to attack the network (the market) itself. Then, attackers can use the fact that sequential computations can be spoiled by just one dishonest move - think of training neural network, when in the last step you reset all weights to zeros and return resulting network. This analysis then suggests impossibly high values of s, making the algorithm of verification impossible to use. Securing networks from this types of attacks has to be addressed in some other ways, and it is quite difficult problem [7].

8 Final notes

In the last section, we proved that the number of steps needed to validate can be, with above assumptions, in the range $1 - 3$. If we think again about the neural networks (a couple of gigabytes per step) use-case, it boils down to what amount of data Provider and Requestor can exchange over the network.
If it is measured in tens of mbytes, then this would seriously prevent such a scheme to be used at all.
If it is measured in hundreds mbytes/gigabytes, then it would be possible to validate small neural networks.
If it is measured in tens of gigabytes, it is possible to validate all currently used networks.

It also is important to note that the number of verification steps cannot be fixed and known to Requestor in advance, because then he would have no incentive to continue behaving honestly after all "save-the-state-for-later" calls were depleted. It should be a random variable. We did not investigate what should be its distribution.

Summing up: thorough the paper, we first presented an use-case for computing market solution and explained verification challenge. In the first part of the paper, we derived black-box algorithm for verification, based on precommitment to intermediate steps of computation. We then explored some of potential attack vectors, including the most complicated to deal with cyclic buffer attack. At the second part, we explored the question posed multiple times in the paper and critical to its implementation, eg. checking how many steps is considered enough. To answer it, we constructed and analyzed simple model of Provider-Requestor interaction, and discovered very counterintuitive fact that the required number of sub-steps to verify is independent of size of task. We tried to see some intuition behind that by thinking of Provider doing work as him investing in solution. After analysis, we've seen that there are some potentially dangerous conditions, in with the algorithm will be impossible to use - namely, high-profit-margins markets. We gave some argument of why this shouldn't be much of a problem in the computing market and finished the analysis with an important remark about the conditions in which the model cannot be used, eg. when there can be other incentives in game, other than simple drive for profit coming from inside the market.

Acknowledgment

Author would like to thank G. Gruszczynski, A. Skrzypczak for their input, in particular for ideas of attacks on the scheme, and L. Glen for his cyclic buffer attack idea.

APPENDIX A

We can remove the need for immutability and the secrecy of salt in the box (opaqueness), at a cost of 2-times increase in computational load of the task. We can remove the requirement of external communication from the provider's machine and have all computation happen there[7]

Now our box B is a function, taking a vector v_i and returning a vector $v_o \in \{0,1\}^k$. It works by finding v_o such that $f(v_o) := \text{Hash}(v_i \text{concat} v_o), f(v_o) < C$ for some predefined value of C (essentially a proof-of-work). Common assumption is that the probability of random $f(v_o) < C$ for a random v_o is $C/2^k$. Another assumptions we will use here will be that:

1. The probability of $f(v_o) > C'$ for some predefined value of C' is $1 - C'/2^k$.
2. The constant C is set such that the cost of calculating B is equal to calculating one step of (original) function f.

The of the task for $\mathbb{R}$ is to receive c saved state transitions from $\mathbb{P}$.

One obvious strategy for $\mathbb{P}$ is to just follow the algorithm:

1. $\mathbb{P}$ calculates the sequence of states (X_t).
2. After every state transition, $\mathbb{P}$ puts the transition (a pair of states) into B.
3. B returns a v_o - and if $f(v_o) > C'$, the state transition should be saved - $\mathbb{P}$ saves it or not depending on the decision.
4. When $\mathbb{P}$ finishes computation, he sends all the hashes and saved states to $\mathbb{R}$, which checks if the number of them equal to what he was expecting, and then proceeds with checking each of saved states.

Is there any other strategy, more computationally efficient, also resulting in c honest state transitions? No. Since the outcome of the decision "save"/"not save" is known only after calculating the state transition, and every state transition will yield "save" with probability $p(= 1 - C'/2^k)$, there is no other way of obtaining c honest state transitions, (which hashes yield "save" decision) then just grind through honest state transitions - and that's exactly what honest strategy is doing.

The problem is with non-deterministic computation. Then, if $\mathbb{P}$ knows the procedure for checking state transition correctness and calculating $f(X'_t)$ where $X'_t = X_t + \epsilon$ is much easier when one knows $f(X_t)$, he can take one honest state transition (X_t, X_{t+1}) and grind through B, changing X_t to X'_t and getting $X'_{t+1} := f(X'_t)$.

[7]It was impossible in the original scheme, because Provider would be able to just snapshot the raw bits of the box and then be able to "remove" something from it just by replacing current copy with a snapshot.

APPENDIX B

We have done some experiments regarding neural networks weights compressing, using standard open-source software (gzip, bzip2, xz, 7zip). It appears that there is very little to gain using compression in such situation.

	uncompressed	gzip	bzip2	xz	7zip
Small network	4.2	3.9	4.0	3.9	3.9
Medium network	168	155	158	153	153
Small network transition	8.5	5.0	5.0	4.9	4.9
Medium network transition	336	309	315	306	305

All sizes are in megabytes.

Appropriate code listing:

```
#!/bin/bash
tar -czvf small.gzip small
tar -cjvf small.bzip small
tar -cJvf small.xz small
7za a small.7z small

tar -czvf medium.gzip medium
tar -cjvf medium.bzip medium
tar -cJvf medium.xz medium
7za a medium.7z medium
```

APPENDIX C

The algorithm described in the paper was implemented as proof-of-concept for hyperparameters optimization for neural network training on GOLEM. It is currently available as pull request to main GOLEM repository, under `https://github.com/golemfactory/golem/pull/1407`. Details on implementation are available under `https://github.com/golemfactory/golem-usecases/blob/machine_learning/README.md`.

Under `https://github.com/golemfactory/golem-usecases/tree/machine_learning/impl`, there is a simple implementation of the protocol, together with some attacks tests, not intended to use as a library, but rather as an example and testing bed.

APPENDIX D

Here, we formalize and explain our try to construct a more general solution for a cyclic buffer problem. It is by no means complete, but maybe it can inspire someone else to build a working one.
We will say that the state X_i is a vector from $\mathbb{R}^s$ space for every i, and that the program is *almost*-deterministic if for any two runs of the program that start from state X_i, corresponding states X_{i+1} and X'_{i+1} differ at most by a factor of $\alpha << 1$. To simplify the argument, let's say that it is equivalent to X_{i+1} and X'_{i+1} be different at at most j least significant bits.
Then, we can introduce a modification to the previously described black box algorithm. Our new black box will take decision whenever a particular state should be saved by looking not only on the hash of (X_k, X_{k+1}), but also on the hash of (Y_k, Y_{k+1}), where Y_l is a rounding of each coordinate of X_l, cutting off j least significant bits.
The new box will take pairs $((X_i, X_{i+1}), (Y_i, Y_{i+1}))$, save them both, and with probability of $\frac{1}{2}$ feed the first item (the non-rounded states) to the original box, and with probability $\frac{1}{2}$ feed the second one. Then, it will return the original box answer.
Then, if the Provider uses cyclic buffer, he will be forced to do either of two things: modify the state strongly (by more than α), because otherwise the box will output the "save" decision exactly in the same steps of cyclic buffer. However, after this modification, theses state transitions are no longer valid, since by assumption, honest tries can differ only by a factor of α, so if the dishonest state is saved, the Requestor is able to spot the cheating. There are several problems with that solution - first, it can end up saving many more states than needed - computation that modifies the state only slightly (by a factor less than α) will always produce the same hash after rounding, so it the Provider will end up saving about $\frac{1}{2}$ of states. Second problem is that Provider can then only modify the states that were saved (since he already knows which one were), so it doesn't really remove the possibility of attack.

References

[1] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533, 2012.

[2] GOLEM. Golem whitepaper, 2017.

[3] Nvidia. Cudnn user guide, 2017.

[4] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[5] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[6] STREAMR. Streamr whitepaper, 2017.

[7] J. Teutsch and C. Reitwiener. A scalable verification solution for blockchains, 2017.

[8] G. Wood. Ethereum: a secure decentralised generalised transaction ledger, 2014.