Both Truebit and proof of delay random beacon are vulnerable to a delay attack.

It consists of an attacker submitting fake answers knowing he will ultimately be disproven, but using it just to delay the system. I'm taking Truebit paper as an example but everything can also be applied to proof of delay (the problem I was initially interested in).

Problem

In the Solver VS Verifiers case, an attacking verifier can challenge the solver multiple times in a row, knowing he will lose, but that each fake challenge will delay the time the result will be available. Since the solver would have to win over all fake challengers, and withstand only one challenge at the same time, this would delay the result by $O(a \frac{log(t)}{log(c)})$

interactions where a

is the amount of attacker fake challenges, t

the amount of computational steps of the computation and c

the amount of checkpoints. This can stall a decentralized application crashing the price of the underlying token. An attacker can simply do that to short the token of the decentralized application affected for profit.

The attacker can even be elected as the solver by paying a bribe b to the miners such that:

$b>r-c$

where r

is the solver reward and c

is the cost of the computation. Honest parties would not have interest to pay a higher bribe as this would mean that the bribe and the cost of computation would be higher than the reward. He can then submit a fake solution, lose his deposit but continue again to bribe the miners such that he is always the solver. This would also lead to a delay of the result of $O(a \frac{log(t)}{log(c)})$

interactions where a is the number of deposit + bribes, the attacker is ready to pay. As in previous attack, it can be made profitable by shorting the underlying token of the application the attacker is delaying. This attack is more costly than the first, but it has the advantage of not letting observers know when it would end, putting more uncertainty on the price of the affected dapps tokens which would profit the attacker (he could also long just after stopping his attack).

We can also see it purely from a griefing perspective and apply the time-discounted utility model. The funds, worth v

, which depends of the computation would have to be frozen for the delay d

. With a discount factor β

, the grief would be $(1-\beta^d) v$

.

To answer challenges, parties won't be able keep all steps in memory, so they will have to keep checkpoint.

Each interaction must have a timout at least high enough to allow going from one checkpoint (in local memory) to another. With a memory m

and checkpoint size s

, party would be able to keep $\frac{m}{s}$

local checkpoints.

Therefore an interaction can take up to $O(t \frac{s}{m})$

. The timeout must be higher than that. So in both cases, the delay would be in $O(a \frac{s}{m} t \frac{log(t)}{log(c)})$

time.

Sketch of solutions

A solution is to let everyone be a solver. If all solvers submit the same answer, it is accepted. If they don't agree, they have to play interactive verification games. If at least one solver is honest, he will always win (and since solvers share the reward, no need of forced errors anymore).

The challenge which comes with it, is that interactive verification was designed for two parties. So we need multiparty interactive verification.

Parallel challenges

One solution could be to run challenges in parallel. If one party must work on multiple verification games at the same time, this party have the time to answer them all.

For example assume that we have 10 000 000 of computational steps and 10 checkpoints. If the value at step 4 999 999 is asked, the party would have to load its 4 000 000 checkpoint and do 999 999 computational steps. This should be possible without being timed-out.

But now if we have 10 challenges in parallel, each asking 999 999, 1 999 999, … 9 999 999 values, the parties would have to do 9 999 990 computational steps and could be timed out. So the timeout between each message of interactive verification should be at least the whole computation time (since we can have more attacker submissions than checkpoints, note that with fewer submissions a smaller timeout can be set, but we are focusing on asymptotic complexity so this would not change O

). So the procedure would take a O(t \frac{log(t)}{log(c)})

time.

Multyparty Verification

One-VS-One

This is the base game between two parties described in both papers.

One-VS-Many

First let’s look at the 1 versus many case.

We want to reduce it to One-VS-One cases.

If the multiple ones always send the same messages in the interactive verification game, it reduces directly to the One-VS-One case.

Else if the multiple parties send different messages, as soon as they do, we pair them against each others to play against each others.

It will take at most O(log(a))

rounds of One-VS-One until only the solo party wins or defeat all the others (because at each round either the solo party lose, or at least half of the multiple parties lose).

Many-VS-Many

Now, we want to reduce the many versus many interactive verification games to multiple one versus many verification games.

To do so we assume everyone starts with 1 point.

Starting with people with the highest amount of points, we assign them opponents such that the total number of points the opponents has is lower than their number of points.

If the solo party wins the game, it wins the points of all others parties. If some of the others parties win the game against the solo party, they double their amount of points.

Since there is always a party (except in last round, or when it is not paired against another party) who will at least wins half of the amount of points it has. We will have a party whose amount of points has an exponential progression. Since no new points are created, this can only last a logarithmic amount of rounds.

Therefore we can have at most O(log(a))

amount of One-Versus-Many rounds.

Overall Complexity

- A one to one interactive verification game takes O(\frac{log(t)}{log(c)})

, interactions.

- A one to many takes O(log(a))

one to one games.

- A many to many verification game takes $O(log(a))$

many to one verification games.

So we can conclude that a many to many verification game takes $O(log^2(a) \frac{log(t)}{log(c)})$

interactions.

It would then take a $O(t \frac{s}{m} log^2(a) \frac{log(t)}{log(c)})$

time.

This should be better than $O(t \frac{log(t)}{log(c)})$

time some applications, in particular for proof of delay where s

is really small.

TL;DR

Interactive verification can let malicious parties delay significantly the result without being able to corrupt it. But we can set it up in such a way that this delay is minimized.