Lookup Singularity via MMR

Introduction

What I will show is that if we have a public merkle mountain range where existence within it implies correctness, then this primitive can be used to achieve the lookup singularity. What I mean is you can enable complex lookup tables for sha256 hash, merklizing leaves, or potentially even EVM storage proofs.

There are optimistic and immediate finality variations of this scheme which balance certain trade offs.

Background

Lookup tables are an incredible innovation in the world of ZKPs. They allow for much faster proving for complex computation like a Sha256 hash. Barry Whitehat coined the term the lookup singularity (Lookup Singularity - General - zk research) to represent how they can drastically improve the performance of SNARKs.

Standard lookup tables require pre-computing an entire table, and committing it during circuit compilation. Therefore these tables cannot be updated after the keys are generated. The cost of committing the table is amortized over all the lookups used during proving. There is a negligible per-lookup cost for these types of constructions (eg plonk-lookup). The obvious limitation here is that it's impractical for lookup tables that are very large, say 32-bit float arithmetic (2^64 rows).

Justin Thaler wrote papers for a new system called Jolt/Lasso which sacrifices the low per-lookup cost in order to get more complex lookups. This is done by using multivariate polynomials to represent the lookup table as a polynomial evaluation. This makes it so you don't need the full table in memory to use in a circuit, but increases the per-lookup cost. See Lasso/src/subtables/lt.rs at 823da601422d17f7c150631947b33a9db1ad5b98 · a16z/Lasso

These constructions don't practically allow for lookup tables of truly complex operations like a sha256 hash. Say I had 100 leaves and I wanted to merklize them to compute the root. No existing lookup table construction can help you with this.

Trusted lookup table oracle

One very easy way to increase the performance and utility of a lookup table is to use a trusted source. Imagine I wanted to multiply 2 floating point numbers, I can submit the lookup via an API to a service that returns the lookup table with a snark-friendly signature. In practice we would batch all the look ups together with one signature.

The beauty of this approach is you can enable truly arbitrary lookup operations, which would dramatically improve the performance of your circuits. You also get a custom lookup table for each proof you generate. This approach would drastically decrease circuit complexity.

One thing to note is that for any sensitive parts of the circuit you may wish to do it without the table oracle to preserve privacy. Alternatively you could include extra unnecessary lookups to obfuscate which were used. This approach lets you offload any parts of the circuit which can be public into a lookup table. Essentially a selectively disclosed computation ZKP.

However, the issue here is that the trusted party can secretly sign incorrect lookups in order to forge a proof and attack a system. The idea I want to present is to solve this malicious lookup oracle problem via crypto-economic methods.

What if there was a better way?

Securing with crypto-economic security

The obvious place to start is have the lookup oracle stake a large amount of ETH/USDC/etc. If I can coerce the oracle to give me a malicious result, I can generate a fraud proof (ZKP or smart contract) in order to claim a reward.

This works well to prevent the oracle from colluding with someone else. This however fails in the situation where the oracle is also the one requesting the lookup. There is no economic incentive to claim your own stake.

The only way to solve this problem is to force the oracle to publicly disclose every lookup table that it generates. This can be done with a merkle mountain range (MMR). The key insight you need to see is that existence within the public MMR directly implies correctness. We then construct the circuits to check for existence within a MMR, and we then compare with the root of the trusted, public MMR.

Overall Solution

First you use a merkle mountain range w/ a zk friendly hash to commit tables into a smart contract. A MMR is advantageous because:

1. Scales virtually infinitely
2. Short inclusion proofs
3. Quick to update proofs

In order to avoid confusion, I want to emphasize the MMR does not

merklize a full lookup table. Instead each leaf in the MMR can be a custom lookup table(s) for a particular proof generated. Each leaf can be multiple tables, as a circuit might offload multiple operations to a table. The MMR may include duplicate lookups.

The lookup table used during proving only needs to include the entries needed. This is the key advantage of this approach. The table(s) can be generated locally by the user before being generating the proof. The tables then can be sent to be added to the global MMR, at which point the proof can be updated to have inclusion in trusted MMR.

To save on gas, the actual lookup tables do not need to be put on-chain, we can use an optimistic approach where just the commitments of the tables are stored. The details around off-chain data availability will not be discussed here.

In situations where we want quick finality, you can have a contract verify table validity before adding it to the global MMR. The trade off is it will cost more. The optimistic approach actually allows for much more complicated look ups (eg call a smart contract w/ some input at a block height)

There are a few variations these tables can be utilized, but the custom lookup table is always an input into the primary circuit. This is nice because in a mobile app setting this enables us to generate a snark and its corresponding tables very quickly. What remains is proving the table used is in the global lookup tree. The mobile app can submit the zkp + lookup table to an infrastructure provider which will finalize the snark.

The infrastructure provider can verify the table validity, and submit it for inclusion into the trusted MMR. Once included, an inclusion proof can be generated. The original snark can be recursively updated to verify the table used exists within the global MMR. The root of the MMR then becomes a public input of the resulting ZKP. The location of the table can optionally be disclosed as well.

The overall trade-off is pretty simple. For the price of waiting a bit to submit/verify the proof, you get fast and memory efficient snarks client-side. The waiting to submit can be handled for the user with infrastructure.

Complex fraud proofs are not required for the optimistic variation. A smart contract can be used to check validity of entries when prompted.

Conclusion

I am very confident the above would work, and yield dramatic performance gains to client-side snark proving.

For the optimistic variation we can enable truly complex lookups that simplify ZKPs around EVM storage proofs. The tradeoff for very complex look ups would be a longer settlement time. The settlement time can be decreased if you limit the lookup operations in complexity (sha256, floating point operations etc).

In the immediate finality variation, there will be a larger financial cost for each addition to the MMR. I suspect the EVM would be prohibitively expensive, especially at scale. To optimize this approach it would be best to build a new chain. As of the time of writing, Solana is doing 3500 TPS at a tx fee around $0.0002. If you reduced the complexity of the execution environment you would get more TPS for cheaper.

Alternative Perspective

It hit me after my initial post that there is an alternative use case for the technique described above where a MMR is unnecessary.

From the perspective of the prover, generating the main ZKP is done in two parts

1. Compute lookup table(s) for circuit
2. Use table(s) as a public input to generate the ZKP

From here we have a ZKP and a list of assumptions the ZKP is made on. The other use case is the prover can offload some of the heavy parts of the circuit to infrastructure to validate the assumptions, and recursively update the ZKP.

Pretty neat