OVERPASS: A SECURE SEARCH DELEGATION STANDARD FOR CHEAPER TRUSTWORTHY COMPUTATION

MU CHANGRUI
JOSHUA NATHANAEL
LIU ZHIYANG
WOO WEN JUN
CHEN KEYING
SARAVANAN ANUJA HARISH

Abstract. When coupled with the theory of computation, the blockchain and cryptographical scheme has demonstrated its utility through several projects, with Ethereum commonly seen as the first decentralized, un-ownable digital computer. In such a decentralized and un-ownable system, control flow integrity is well guaranteed, but the execution of logic and change of states in a such consented virtual machine is slow and expensive due to the message-passing and state consensus between physical nodes.
OverPass provides a standard for search delegation where an untrusted advisor gives hints/answer/proof to the client on-chain and extends the computational power of address-based consented virtual machine on solving interactive proof (IP) problems.

1. Introduction

Smart contracts introduced by Ethereum [**wood2014ethereum**] provide trust computing. Once a smart contract has been engaged, it will complete exactly how it was coded and no parties can interfere or change the result. However, the computation is usually expensive and slow. There exists a set of problems where there is signs showing that verifying answers is cheaper than searching for the answer. Hence well-defined incentive mechanism can be applied to delegate searching computation to untrusted computational power (off-chain server) while the completeness and soundness of the computation are still guaranteed:

- A delegation task is posted on smart contract together with a trusted verification function.
- Some untrusted machines would listen to the task event and provide answers/advice that could help smart contract solve the problem.
- With the hints from advisor, the client on-chain can compute the legitimate answer in a cheaper way.
- If the hints is illegitimate given by a dishonest advisor, the client will accept with negligible probability.

1.1. Driving Factor.

The project is driven by the Pros and Cons own by traditional computer and consented decentralized computers as shown in the following table:

Function	Pros	Cons
Traditional Computer	fast and cheap	un-trusted
Consented Computer	trusted	slow, expensive

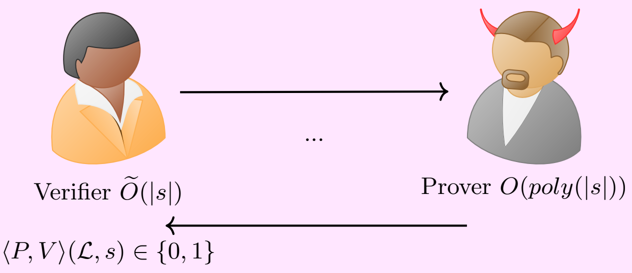
OverPass works as a standard to balance the Pros of these two and achieve cheaper and faster trustworthy computation.

1.2. Theoretical Background.

The theory of computation is shaped by the Interactive Proof (IP) system[**GMR**], where a strong, possibly malicious, prover interact with a weak verifier, and at the end of the computation, the client can output an answer achieving completeness and soundness. There are signs that many problems has cheaper verification algorithm than search algorithm(e.g., NP-complete, sorting). This triggered a novel idea on trust-worthy computation that not all work should be done on the trusted slow "computer", or verifier, as long as the un-trusted computational power can provide proof for the answer to the verifier. This would expand the computational power of consented computers (e.g. EVM) tremendously while the trust of the computation is maintained. The class of problem we focus are problems with doubly efficient IP scheme[**ip_muggles**][**goldreich_doubly_efficient_ip**] and some P problems with cheap verification system than existing search algorithm.

Doubly Efficient Interactive Proof:

$s \in \{0,1\}^*$, Language $\mathcal{L}$


...
Verifier $\widetilde{O}(|s|)$
Prover $O(poly(|s|))$
$\langle P, V\rangle(\mathcal{L}, s) \in \{0,1\}$

- Completeness: $Pr[\langle P, V\rangle(\mathcal{L}, s) = 1 | s \in \mathcal{L}] \geq 1 - negl_1(|s|)$
- Soundness: $s \notin \mathcal{L} : \forall P^* : Pr[\langle P^*, V\rangle(\mathcal{L}, s) = 1] \leq negl_2(|s|)$

An incentive mechanism is made such that miners have the incentive to mine by executing the protocol honestly and the verifier has the incentive to participate and save gas fees from achieving the same goal. An example of the incentive mechanism is as follows:

client \advisor	Honest	Cheat
use OverPass	(a - gas_{vrfy}-i, i- $gas_v rfy$)	(0, - gas_{vrfy})
not user OverPass	(a- gas_{search}, 0)	(a - gas_{search}, 0)

- a: the incentive got by user for getting the legitimate answer of the problem.
- i: the incentive paid by user to advisor.
- gas_{vrfy}: the gas fee for verifying an answer.
- gas_{search}: the gas fee for searching an answer.

Given that gas for searching is much higher than gas for verifier, and a wise incentive is set by the client, the (client use OverPass,Advisor be honest) is the Nash Equilibrium. Note that the mechanism would work under the assumption that the client is a rationale and at least one advisor is rationale and selfish, which is very robust.

1.3. **Previous Work.** There are currently products also serve on improving the price and speed of trustworthy computation, which are Solana and Arbitrum. First, Solana is a layer 1 blockchain that uses Proof of Stake (PoS) with Proof of History (PoH) to accelerate its average block validation times, which is around 10 times faster than Ethereum [**tyson_2022**]. It also has a lower gas fee per transaction, which is around $0,0000014 per transaction [**beincrypto_2022**]. Second, Arbitrum is a layer 2 blockchain that is built on top of Ethereum. It uses Interactive Fraud Proofs, which is a type of Interactive Proofing. The basic idea of Interactive Fraud Proofing is that both the prover and verifier will try to check/bisect each other answers until they disagree to one or more things. This will, then, be run on the main Ethereum network. On the other hand, our project will be focusing on solving problems with cheap NP proof or doubly efficient interactive proofing, as shown in Section 1.2.

2. OverPass Design

The protocol is named OverPass as it is aimed to serve as an Overpass Bridge across computationally expensive tasks for Users in a trustworthy manner with lighter Gas cost.

2.1. **Legal Roles.** in the OverPass Protocol include Users, Miners, and OverPass instances. An OverPass instance is a Smart Contract instance that follows the OverPass Protocol to provide a platform for Users and Miners to interact regarding a particular Question type, this is referred to as the Kernel Question of an OverPass instance. Users will post computational tasks (of the Kernel Question Type) with a reward through the OverPass instance. Miners will be any Computation Providers that listen to computational tasks posted by Users through the OverPass Contract and cast Advice (unverified solutions) in exchange for incentives. Multiple OverPass contracts of different Kernel Questions can exist on the chain, and any User can post respect question instances to the Contract Instances, while any Miner can listen and cast solutions.

2.2. **The Trust.** in OverPass Protocol is built by assuming the rationality of the Miners: Casting advice cost Gas due to casting itself needs consensus and running of the Verification function of the OverPass Contract but only the optimal answer provider will be rewarded. Computing the correct answer costs little locally and casting a wrong / non-optimal answer wont cost much less. Therefore rational miners would provide the best possible advice.

2.3. **The Flow.** of the OverPass Protocol is visualized in Figure 1. First, a User calls delegateCompute method with a Question, Reward and Time limit. This Commit will cause the reward to be granted to the first Miner who provided the best Advice within the Time limit. The contact emits an event for any new question posted to inform the Miners who are listing to the Contract address. A miner who is satisfied with the potential reward then casts an answer by calling the Advice method with proof and solution and the Verify method will check the validity and optimality before keeping the new answer as the best answer.

Once the lifetime of a question has ended, the User can retrieve the best answer so far by calling the getAnswer method with the respective taskId. And the miner can call getIncentive to check if his answer is accepted and obtain a reward if so.

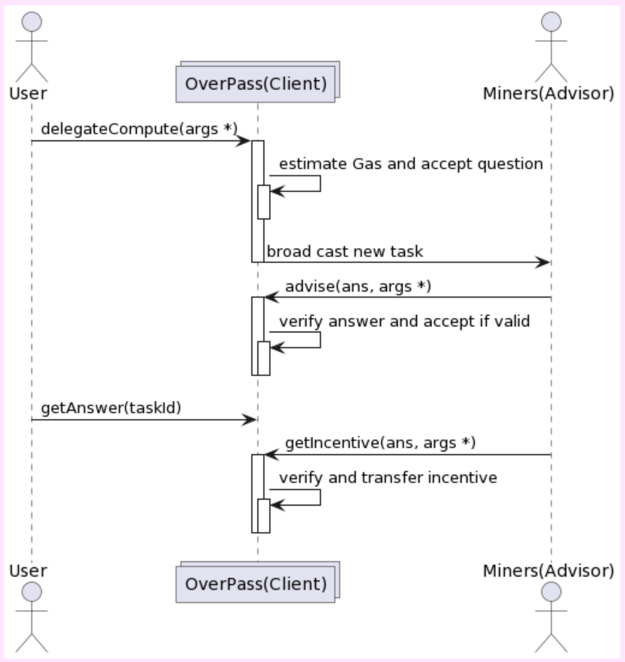


Figure 1. OverPass Protocol Flow

3. Demonstration / Data Collection

The demonstration is done with Longest Common Subsequence(LCS)[**holmgren_et_al:LIPIcs.ICALP.2022.73**] being the Kernel Question. The best search algorithm has a time complexity of O(m*n), where m, n are the length of the two sequences, but Verifying, if a given Sequence is indeed a common Subsequence of the two, takes only O(n), n being the shorter length of the two Sequences.

For this Kernel Question, a miner must submit the length of the LCS and the LCS itself as proof for verification.

In the demonstration, an LCS Overpass instance is first deployed on the Ganache Test Network as shown in Figure 2. To showcase the superiority of the OverPass Protocol, the deployed instance also includes a compute method that computes the LCS problem fully on EVM(Figure 3).

Figure 2 Terminal 1: Testnet

```
> cd testnet
> sh start_gananche_testnet.sh 100
```

Figure 3 Terminal 2: LCS

```
> python3 demo.py LCS
contract address: 0xe78A...
times_to_compute: > 20
```

Figure 4 Terminal 3: LCSOverPass

```
> python3 demo.py LCSOverPass
contract address: 0xe92A...
times_to_delegate: > 20
```

Figure 5 Terminal 3: LCSOverPass

```
> python3 miner
Available orders:
1. listen <contract_address>
2. unlisten <contract_address>
3. min_incentive <min_incentive>
4. maximum_duration <maximum_duration>
5. get_incentive
> listen 0xe92A...
> get_incentive
```

Then, a User instance is created and many deterministic random test cases of different lengths and alphabet sets are generated and fed into the LCS OverPass instance with both delegateCompute (Figure 4)and compute(Figure 3). The average Gas cost for the test cases computed in two different ways is recorded.

And, of course, a Miner instance(Figure 5) is created and to listen to the LCS Overpass Contract address and will provide advice for any Tasks whose reward exceeds a fixed number. More detailed instruction introduction can be found in [**mu_nathanael_liu_woo_chen_harish_2022**].

4. Result Analysis

We perform computation of LCS over 1000 pairs of random strings and the result shows that the gas fee before and after adopting our overpass protocol will be discussed. We use the same gas fee as Ethereum (0.09$) for estimation. The following table shows the average gas per LCS test case.

	Original	Overpass
gas	16,990,800	3,538,362
gas fee(Wei)	1.86 e+17	3.54 e+16

As shown in the table, the gas consumption is 16,990,800 before applying overpass. After optimization, it only costs about 3,538,362 which is an 80% reduction in the gas fee. This result indicates that our overpass protocol optimizes smart contracts by providing cheaper gas consumption for complex computation problems. (The gas fee is estimated based on the Ethereum gas price in November 2022).

The reduction in gas fees is the value our protocol can make. Part of it will be rewarded to miners as incentives to carry out computation/transaction property and motivate them for future execution. It is also beneficial to maintain a decentralized system.

Our protocol results in win-win cooperation where users reduce costs and miners receive incentives.

5. Conclusion

Overpass is a standard for faster and cheaper trustworthy computation on smart contracts. The correctness of execution is guaranteed by the verification algorithm on the smart contracts. Off-chain servers are incentivized if the answer passes verification. An equilibrium is achieved where miners receive incentives by reducing users costs under the trust of smart contracts. With OverPass standard, problems with cheap NP verification or doubly effective interactive proof can be computed in a cheaper and faster manner, and this contributes to moving consented trustworthy computers towards a faster, cheaper, and stronger era.

Appendix A. Overpass Standard

```
// contracts/OverPass.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

abstract contract OverPass {

    struct Task {
        address taskOwner;
        uint256 taskId;
        string [] taskParameters;
        address bestAdvisor;
        uint bestAnswer;
        uint incentive;
        uint approxGasFee;
        uint startBlock;
        uint computePeriod;
    }


    event postNewQuestion(uint256 taskId, uint incentive, uint approxGasFee, uint computePeriod, string[] taskParameters);
    event updateQuestionAnswer(uint256 taskId, uint bestAnswer, address bestAdvisor);
    event complateQuestion(uint256 taskId, uint bestAnswer, address bestAdvisor);


    // Store the information of tasks
    mapping(uint256=>Task) internal tasks;

    uint internal nonce=1;

    // First eight hex of keccak-256(problemName(<problem input data types>))
    bytes problemSig;

    constructor(bytes memory _problemSig) {
        require(_problemSig.length==8, "Illegal length of problemSig");
        problemSig = _problemSig;

    }

    modifier winnerOnly(uint256 _taskId) {
        require(
            block.number>tasks[_taskId].startBlock+tasks[_taskId].computePeriod&&msg.sender == tasks[_taskId].bestAdvisor ,
            "winner only function"
        );
        _;
    }



    // Called by user to delegate compute specific problem
    function delegate_compute(string[] memory taskParameters, uint _computePeriod) payable public virtual returns (uint256);

    // Called by miner to advise hint/answer/proof
    function advise(uint256 taskId, uint256 ans, string[] memory proof) payable public virtual;

    // Called by winner miner to get the incentive
    function getIncentive(uint256 _taskId) public virtual;
        function getTaskIncentive(uint256 taskId) public view returns(uint) {
        return tasks[taskId].incentive;
    }

    // Called by miner to get the estimated gas fee
    function getTaskApproxGasFee(uint256 taskId) public view returns(uint){
        return tasks[taskId].approxGasFee;
    }

    // Called by miner return parameters of a function
    function getTaskParameters(uint256 taskId) public view returns (string[] memory){
        return tasks[taskId].taskParameters;
    }
}
```

Appendix B. LCSOverPass instance

```
// contracts/OverPass.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;
import "./overpass.sol";


contract LCSOverPass is OverPass{
    // 6c633bc4 is first eight hex of keccak-256(problemName(lcs(string,string))
    constructor() OverPass(("6c633bc4")) {
    }

    mapping(uint=>uint) answers;


    // simulate the verification to estimate the gas fee
    function _simulateVerification(bytes memory s1,  bytes memory s2, bytes memory s3, uint k) private pure
         returns (bool){
        uint checkI = 0;
        // simulate length check
        if (s3.length != k) {
            return false;
        }
        // simulate check over s1
        for (uint i=0; i<s1.length; i++) {
            if (checkI==s1.length) {
                break;
            }
            if (s1[i]==s1[checkI]) {
                checkI +=1;
            } else {
                continue;
            }
        }
        if (checkI!=s1.length) {
            return false;
        }
        checkI=0;
        // simulate check over s2
        for (uint i=0; i<s2.length; i++) {
            if (checkI==s2.length) {
                break;
            }
            if (s2[i]==s2[checkI]) {
                checkI +=1;
            } else {
                continue;
            }
        }

        if (checkI!=s2.length) {
            return false;
        }
        return true;
    }

    function _verifyResult(bytes memory s1,  bytes memory s2, bytes memory lcs, uint k) private pure returns
          (bool){
        require(lcs.length==k, "proof and answers are unmatched.");
        uint checkI = 0;
        for (uint i=0; i<s1.length; i++) {
            if (s1[i]==lcs[checkI]) {
                checkI +=1;
                if (checkI==k) {
                    break;
                }
            }
        }
        if (checkI!=k) {
            return false;
        }
        checkI = 0;
        for (uint i=0; i<s2.length; i++) {
            if (s2[i]==lcs[checkI]) {
                checkI +=1;
                if (checkI==k) {
                    break;
                }
            }
        }
        if (checkI!=k) {
            return false;
        }
        return true;
    }

    uint MAX_STR_LEN = 1001;
    // called by other smart contract to compute specified algorithm with parameters
    function delegate_compute(string[] memory taskParameters, uint _computePeriod) payable public override returns (uint256) {
        bytes memory _problemSig =  bytes(taskParameters[0]);
        require(taskParameters.length==3 && _problemSig.length==problemSig.length && keccak256(_problemSig)==keccak256(problemSig), "Wrong delegated task.");
        bytes memory s1  = bytes(taskParameters[1]);
        bytes memory s2 = bytes(taskParameters[2]);
        require(s1.length<MAX_STR_LEN && s2.length<MAX_STR_LEN, "String too long.");
        _simulateVerification(s1, s2, s1, s1.length);
        require(msg.value>tx.gasprice*10, "not enough payment.");
        // set new task

        tasks[nonce] = Task(msg.sender, nonce, taskParameters, msg.sender, 0,  msg.value, tx.gasprice, block.number, _computePeriod);

        emit postNewQuestion(/*taskId*/nonce, /*incentive*/msg.value, /*approxGasFee*/tx.gasprice, _computePeriod, taskParameters);
        nonce += 1;
        return nonce-1;
    }



    function advise(uint256 taskId, uint256 ans, string[] memory proof) payable public override {
        require(tasks[taskId].taskId>0, "task not exist");
        require(ans>tasks[taskId].bestAnswer, "Better solution exists");
        require(proof.length==1, "Invalid proof");
        bytes memory s1 = bytes(tasks[taskId].taskParameters[1]);
        bytes memory s2 = bytes(tasks[taskId].taskParameters[2]);

        bool isValidAns = _verifyResult(s1, s2, bytes(proof[0]), ans);
        require(isValidAns, "Invalid proof");
        tasks[taskId].bestAdvisor = msg.sender;
        tasks[taskId].bestAnswer = ans;
        answers[taskId] = ans;
        emit updateQuestionAnswer(taskId, ans, msg.sender);

    }

    function getIncentive(uint256 _taskId) public override winnerOnly(_taskId) {
        uint incentive = tasks[_taskId].incentive;
        (bool success, )  = msg.sender.call{value: incentive}("");
        require(success, "Failed to transfer the incentive");
        delete tasks[_taskId];
        emit complateQuestion(_taskId, tasks[_taskId].bestAnswer, tasks[_taskId].bestAdvisor);
    }

    function checkAns(uint256 taskId) public view returns (uint) {
        return answers[taskId];
    }


}
```

Appendix C. A LCS algorithm

```
// contracts/OverPass.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

contract LCS {

    function lcs(string memory _XS, string memory _YS) public pure returns (uint256){
        bytes memory _X= bytes(_XS);
        bytes memory _Y= bytes(_YS);


        uint256 r = bytes(_X).length+1;
        uint256 c = bytes(_Y).length+1;

        uint[] memory dp = new uint[](r*c);
         for (uint i = 1; i < r; i++){
            for (uint j = 1; j < c; j++){
                if (_X[i-1]==_Y[j-1]){
                    // array[i][j] = array[i-1][j-1] + 1;
                    dp[i*c+j] = dp[(i-1)*c+j-1] + 1;
                }
                else {
                    dp[i*c+j] = dp[(i-1)*c+j]>dp[i*c+j-1]? dp[(i-1)*c+j]:dp[i*c+j-1];
                }
            }
        }
        return dp[(r-1)*c+c-1];
    }
}
```