OpenSquare: Decentralized Repeated Modular Squaring Service

Sri Aravinda Krishnan Thyagarajan
Friedrich Alexander Universität Erlangen-Nürnberg
Nürnberg, Germany
t.srikrishnan@gmail.com

Tiantian Gong
Purdue University
West Lafayette, USA
gong146@purdue.edu

Adithya Bhat
Purdue University
West Lafayette, USA
abhatk@purdue.edu

Aniket Kate
Purdue University
West Lafayette, USA
aniket@purdue.edu

Dominique Schröder
Friedrich Alexander Universität Erlangen-Nürnberg
Nürnberg, Germany
dominique.schroeder@fau.de

ABSTRACT

Repeated Modular Squaring is a versatile computational operation that has led to practical constructions of timed-cryptographic primitives like time-lock puzzles (TLP) and verifiable delay functions (VDF) that have a fast growing list of applications. While there is a huge interest for timed-cryptographic primitives in the blockchains area, we find two real-world concerns that need immediate attention towards their large-scale practical adoption: Firstly, the requirement to constantly perform computations seems unrealistic for most of the users. Secondly, choosing the parameters for the bound **T** seems complicated due to the lack of heuristics and experience.

We present OpenSquare, a decentralized repeated modular squaring service, that overcomes the above concerns. OpenSquare lets clients outsource their repeated modular squaring computation via smart contracts to any computationally powerful servers that offer computational services for rewards in an unlinkable manner.

OpenSquare naturally gives us publicly computable heuristics about a pre-specified number (**T**) and the corresponding reward amounts of repeated squarings necessary for a time period. Moreover, OpenSquare rewards multiple servers for a single request, in a sybil resistant manner to incentivise maximum server participation and is therefore resistant to censorship and single-points-of failures. We give game-theoretic analysis to support the mechanism design of OpenSquare: (1) incentivises servers to stay available with their services, (2) minimizes the cost of outsourcing for the client, and (3) ensures the client receives the valid computational result with high probability. To demonstrate practicality, we also implement OpenSquare's smart contract in Solidity and report the gas costs for all of its functions. Our results show that the on-chain computational costs for both the clients and the servers are quite low, and therefore feasible for practical deployments and usage.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484809

CCS CONCEPTS

• **Security and privacy** → *Cryptography*.

KEYWORDS

Repeated modular squaring; Smart contracts

ACM Reference Format:
Sri Aravinda Krishnan Thyagarajan, Tiantian Gong, Adithya Bhat, Aniket Kate, and Dominique Schröder. 2021. OpenSquare: Decentralized Repeated Modular Squaring Service. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3460120.3484809

1 INTRODUCTION

Time-lock puzzles (TLPs) [36] and verifiable delay functions (VDFs) [8] are two closely related cryptographic primitives that rely on enforcing computational delays to achieve their respective functionalities. TLPs [36] allow embedding of messages inside puzzles "to the future" and they express the notion of time by enforcing a pre-defined number of sequential computation steps to solve the puzzle and access the "future message". VDFs [8] lets users evaluate a function such that the evaluation requires a pre-defined number of sequential computation to all the parties, while offering an easy way to verify the correctness of the function output. Both primitives have numerous applications: Timed primitives, such as homomorphic TLPs [27], timed commitments [9], and timed signatures [9] enjoy a wide range of applications: fair contract signing [9], sealed-bid auctions [9], zero-knowledge arguments [17], non-malleable commitments [26], timed payments in cryptocurrencies [40, 41], among many others [24]. VDFs have found themselves useful in generating randomness beacons [31, 12, 39] which are required in Proof of stake based blockchain consensus [15, 31]. All practical constructions of TLPs [36, 27] and VDFs [44, 35] use the sequential nature of repeated modular squaring in the RSA group for enforcing the computational delay.

Repeated Modular Squaring. The operation of repeated modular squaring in the RSA group can be described by giving $(g, \mathbf{T}, N)$,

where $N = pq$ (for some large primes p and q) is the RSA modulus, $\mathbf{T}$ is some positive integer, and g is some random element in the group $\mathbb{Z}_N^*$. The task is to compute $g^{2^{\mathbf{T}}} \bmod N$. Rivest, Shamir, and Wagner [36] conjectured that the only way to complete this task is to perform $\mathbf{T}$ number of repeated squaring $g, g^2, g^4, g^8, \ldots, g^{2^{\mathbf{T}}}$. This is despite an adversary having any amount of parallel computing power. There is a tremendous amount of interest in developing hardware with specific enhancements for repeated modular squaring [1] to deploy these TLP and VDF constructions in practice. Most of these applications require users to compute the squaring locally in the background actively.

Real-world Challenges. However, there are some key real-world challenges regarding the repeated squaring computation that have remained unanswered to a large extent.

Challenge I: Performing computations constantly in the background seems to be an issue for most users with moderate computing machines. Investing on and maintaining local hardware could be wasteful specially if the usage remains infrequent. The only remaining choice is to outsource this computation to some computing server (that may possess specialized hardware) at some cost. However, this comes with the following problems: (a) in case of time-lock puzzles, users may not want the computing server to learn any information about the puzzle itself, other than simply performing the squarings, (b) users require the guarantee that they receive the correct result of the computation within some reasonable time, or else the computing server is not paid and (c) users want the outsourcing market to be live and liquid so that some servers are available to perform the required number of squarings.

Challenge II: Security of these applications crucially relies on the ability of honest parties to accurately predict the number of squarings $\mathbf{T}$ corresponding to some real-world delay, both for themselves and also for the adversary. Disparity in hardware availability, running costs and geo-political factors, among users makes this prediction much harder. This is one of the major practical limitations of using sequential squaring based TLPs and VDFs [43]. While there has been interesting theoretical lower-bounds available [45], and bounties and competitions are happening, we require real-world estimates that reflect the ever-changing hardware market.

In this work, we strive to solve both challenges simultaneously. Our solution is to have a public outsourcing mechanism where users, whom we refer to as *clients* from now on, can request a pool of servers (possibly with specialized hardware) that offer their computational services. Specifically, a client requests services to perform $\mathbf{T}$ number of repeated modular squarings, in exchange for some payment. By public, we mean that the request and the computational result from the services are available on a public ledger. This mimics an open market place where the requests get matched with results and one can openly learn the relation between $\mathbf{T}$, real time taken to post the computation result and the price offered in the request. Moreover, this knowledge also evolves with the evolving hardware and can be used to make reasonable estimates on $\mathbf{T}$ for other applications of TLPs and VDFs.

1.1 Our Contribution

We can summarise our contribution as follows.

(1) We propose OpenSquare, a decentralized repeated modular squaring service protocol (Section 3), that lets users (clients) request a decentralized network of services (servers) to perform repeated modular squarings, and get paid a reward for the computation. More specifically, clients post a request to a smart contract on a blockchain, and services respond to the contract with their results. The contract is entrusted with paying the services for a correct result. We formally model our system as a decentralized solution service (Appendix B) in the UC-framework [13]. Our protocol is (a) Sybil resistant, where servers cannot duplicate themselves to gain more reward, (b) optimistically efficient, where if everyone behaves honestly, the on-chain computational cost is minimized and (c) incentive-compatible, where it is profitable for servers to offer repeated modular squaring services.

Using our protocol, clients can outsource the solving of their TLPs to services with special hardware in an unlinkable manner, where the services cannot link the request to a specific time-lock puzzle. The decentralized nature means that if some machine crashes, the client can be assured that many other services are working on his request, who would give him the correct computation result. Also, our protocol can help servers profit by using their otherwise idle resources (special hardware) to offer services to clients for payment and profit. As hinted above, our protocol results in an evolving public marketplace. We can make reasonable estimates about the choice of $\mathbf{T}$ in all the applications of TLPs and VDFs.

(2) We give game-theoretic analysis (Section 4) to back the design of our protocol. More precisely, we analyze reward distribution functions for our protocol that balances the following two goals: (a) incentivize servers to stay live and offer their computational services, and (b) minimize the cost for the client who outsources the computation. Additionally, it achieves *dominant-strategy incentive compatibility (DSIC)*, tractable in polynomial time, and (c) guarantees that the client indeed obtains the correct computational result with high probability.

(3) We implement the smart contract of our OpenSquare protocol (Section 5). We demonstrate the gas costs for various functions of the contract and show the practicality of our system. We incorporate implementation level optimisations for certain functions of our contract that helps us save upto 75% of the gas costs for the users using the contract. These optimisations may also be of independent interest for other applications. Given the applications of repeated modular squaring in the form of VDFs in Ethereum 2.0, it is also possible that the system reduces gas costs for VDF related on-chain computation, which further lowers the costs of OpenSquare.

Consequences. Practical privacy-preserving cryptocurrency timed payments were implemented in [40, 41], which opens the doors for several new practical functionalities, like payment channels, multi-signature transactions, atomic swaps etc., to be built on currencies like Bitcoin, Ethereum, and Monero, among many others. As these timed payments rely on users needing to continuously solve time-lock puzzles [27], they can use OpenSquare to outsource their repeated squaring operation and then solve the puzzles. Previously in [40, 41], users could only participate in as many protocol instances (like, payment channels, atomic swaps, etc.) as the number of CPU cores they possessed, as one core was used in solving one puzzle. Apart from alleviating computational effort, another advantage of using OpenSquare to solve puzzles is that users can now

participate in as many instances of these applications as they want, since they are not limited by their own computational resources anymore.

1.2 Solution Overview

We give a brief overview of our OpenSquare protocol. We rely on a publicly verifiable blockchain like Ethereum that offers support for running smart contracts. On a high-level, we implement a contract C that has interfaces using which clients can post computation requests and servers can post corresponding solutions. The contract is also responsible for transferring payments from the client to the servers if the server has posted a valid solution. In the description below, we start with a straw-man proposal and highlight various problems with this simplistic approach, and briefly sketch how we can fix each of those.

The client starts off by posting a request of the form $(g, \mathbf{T}, N)$ to the contract. The client also offers a reward of p coins for the first valid solution. The servers (or services) now have the duration t_{sol} of the solving phase, to post their solution $y := g^{2^{\mathbf{T}}} \bmod N$. The servers also attach a proof of correct computation π using the constructions from [44, 35]. This proof helps in verifying if y is computed correctly without actually performing the repeated squarings again. The contract rewards the service that posted the first valid solution with the p coins. If no solution was posted until t_{sol}, the p coins are refunded back to the client.

Incentive Level. The straw-man proposal suffers from several glaring issues at various levels of the protocol. At the incentive level, given that the computation is deterministic in nature, it is guaranteed that irrespective of parallel computation power, the hardware with the fastest clock cycle (and therefore the fastest single squaring) computes $g^{2^{\mathbf{T}}} \bmod N$ first, and gets the reward. This discourages marginal servers with powerful yet not the fastest machines from participating, and over time they tend to exit the outsource market, leading to centralisation in the market.

We can rectify this by letting the client reward the server with the solution that costs the least, rather than the server with the fastest solution. We realize this approach by requiring the servers commit to an asking price along with their solution: A plain revealing of the asking price would lead to other servers quoting related and favorable asking prices. For instance, if server A quotes an asking price of 5 ETH in plain, server B whose original asking price would have been 5.1 ETH, could quote an asking price of 4.999 ETH and get the total reward amount p for a negligible loss in payment.

Therefore, only after the solving phase is over, the services reveal the asking price a, which the services expect from the total reward amount p. The contract rewards the server with the least asking price by transferring the asking price from the total reward budget p to the server. While this mitigates the issue of the fastest machine always wins, the single service reward is still an issue for the decentralization of the marketplace.

We can address this problem by letting the client specify a reward distribution function $\mathcal{R}$ that, depending on the asking prices of different services and the total reward amount p, determines the reward price for k services, instead of just rewarding one service. In Section 4 we formally analyze the requirements of such a reward distribution function to incentivize a maximum number of servers to stay in the outsourcing market in the long run.

Cryptographic Level. Note that since the computation is deterministic, the solution $g^{2^{\mathbf{T}}} \bmod N$ is uniquely determined by the request $(g, \mathbf{T}, N)$. Therefore, the servers in the network can copy the solutions of other servers and register them at the contract. In this case, the servers copying the solutions of others have a good chance of getting the reward price for free, i.e., without investing any computational cost. Also, an issue with rewarding multiple solutions (as discussed above) is a version of a *Sybil attack*, where a server can copy its solution several times in the hope of gaining more reward prices at the cost of computing just once.

To prevent the former attack scenario, we need the property of *non-transferability* of solutions, which says that another server cannot copy the solution of one server without having spent at least $\mathbf{T}$ amount of computation steps. A simple workaround would be that the services commit to their solutions and reveal the solutions later as they do for the asking price. This approach, however, means that the client has to wait until the commit phase or solving phase is over to learn a valid solution, which is undesirable. Instead, our goal is that the client learns the solution as quickly as possible, i.e., after any service registers the solution to the contract.

We can address this issue by using watermarked VDFs [44], where the proof of correct computation π for the solution is watermarked with the public key pk of the server computing it. The watermarking property ensures that no other server with a different public key can compute the proof π in a significantly fewer number of steps than $\mathbf{T}$. This is because the proof π computation requires the knowledge of the intermediate square values between g and $g^{2^{\mathbf{T}}}$. By the security of the VDF construction [44], the intermediate values can only be computed via repeated squaring starting from g.

For the Sybil attack scenario, where the server copies its solution, the above watermarked VDF solution seems insufficient. This is because a server can honestly compute (y, π) watermarked with key pk as above, and compute (y, π_1) watermarked with pk_1, compute (y, π_2) watermarked with pk_2, and so on. The server can do this without spending $\mathbf{T}$ steps for every solution because the server can locally store the required intermediate square values while computing (y, π) and reuse them for computing π_1, π_2, etc. We can circumvent this problem by letting the servers perform $\mathbf{T}$ repeated modular squaring starting from another element x, such that $x := H(g, pk)$ where H is a hash function and pk is the public key of the server. The server attaches a proof of correct computation for $x^{2^{\mathbf{T}}} \bmod N$ as well. In this case, for the server to copy its solution for different public keys, the server has to perform $\mathbf{T}$ repeated squarings for different starting points corresponding to each of those public key, and this cannot be done in less than $\mathbf{T}$ steps for any of the keys.

On-chain Level. Even with the smart contract support, running on-chain cryptographic computation can be pretty expensive; for example, gas costs for group operations in the RSA group in Ethereum are already several hundred dollars at the time of writing this paper. Therefore, the cost of verifying the proof of correct computation π at the contract for every solution being posted can be prohibitively high. This makes the protocol unusable and clogs the blockchain

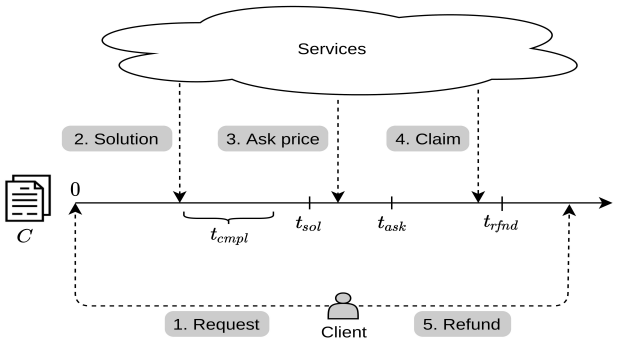

Figure 1: A sketch of OpenSquare protocol. The steps of the protocol are shown in order where each step is an invocation of the contract C. Starting with (1) the client posting a request to C, (2) services posting their solutions to C, (3) later revealing the asking prices for their solutions, (4) and claim the reward, and finally (5) the client refunding the coins from the contract. The complaint step (not shown) is performed during the time interval t_{cmpl} in rare cases where invalid solution is registered. We have $t_{sol} + t_{cmpl} < t_{ask}$.

network with on-chain verification of the proof of every incoming solution.

We solve this problem by letting the contract assume that, by default, every solution is valid. The servers still post the proof π along with their computation result $y_1 := g^{2^{\mathbf{T}}} \bmod N$ and $y_1 := x^{2^{\mathbf{T}}} \bmod N$, but the contract does not verify the proof. Instead, users in the network may verify the proof π locally and complain to the contract if the proof is invalid. This can be done by any other user in the network within t_{cmpl} amount of time after the solution was posted to the contract. If a complaint is registered against a solution, the contract then verifies the proof, and if the proof is indeed invalid, the solution is marked as invalid and cannot receive any reward price.

The above solution is still not complete, as someone still has to pay for the verification in the contract for a complaint, and there is still the possibility of spamming with false complaints. We solve both issues by letting the complainant lock a complaint fee V_c to the contract while registering a complaint. The on-chain cost of verification is paid from the fee V_c irrespective of whether the complaint is valid or is spam. This discourages users from spamming the contract with complaints. On the other hand, to incentivize valid complaints, we ask the servers to lock a collateral amount V_s along with their solutions, which will be transferred to the complainant if his complaint against the solution is valid. If the server's solution is invalid and a complaint is registered against this solution, the complainant is reimbursed with the V_s coins of the server that was locked as collateral. The overall effect of the above two measures is that, only in cases where invalid solutions are registered, on-chain cryptographic computation is necessary, and the cost is covered in an incentive-compatible way.

Putting things together - OpenSquare. A sketch of our OpenSquare protocol is given in Figure 1. The client has $(g, \mathbf{T}, N)$ (where N is a RSA modulus) and posts a request to the contract C (Step 1) in the form of a transaction containing

$$((g, \mathbf{T}, N), t_{sol}, t_{Rfnd}, t_{cmpl}, t_{ask}, k, (p, \mathcal{R}), V_s, V_c)$$

that contains all the relevant information regarding the modular squaring problem, the reward distribution $\mathcal{R}$ and the duration of different phases of the contract. The transaction also locks p amount of coins as the reward amount.

The solving phase t_{sol} immediately follows once the request has been registered at the contract. A service with public key pk, has time until t_{sol} to register the solution

$$\left(y_1 = g^{2^{\mathbf{T}}}, y_2 = x^{2^{\mathbf{T}}}, \pi, pk\right)$$

at the contract, where $x := H(g, pk)$ (step 2). The solution also has a commitment h to an asking price, and locks V_s amount of coins as collateral of the service. The service is guaranteed to receive back the collateral later if no valid complaint is made against its solution.

Every party has until t_{cmpl} amount of time after a solution is registered, to complain about the solution. The complaint also pays a complain fee, V_c amount of coins to the contract. If the complaint is valid (i.e., the solution is wrong), the collateral amount V_s of the corresponding service is transferred to the complainant. On the other hand, if the solution is valid, no action is taken.

After the solving phase is over, the services have until t_{ask} amount of time to reveal the asking price of their solutions (step 3). It is important to note that parties only reveal their asking price after the solving phase is over. This prevents services from registering solutions and commitments to asking prices, depending on the asking price of other services.

Once the asking phase is over, services can claim their rewards for posting a valid solution (step 4). The contract determines the reward p^* for the service, based on the reward distribution $\mathcal{R}$ and the asking price of the service. The contract returns the reward p^* and the collateral V_s locked by the service to the service. The client can recollect any of the remaining coins locked in the contract after time t_{Rfnd} has passed (step 5). We model this work flow through a procurement auction with entry costs. Essentially it's a combinatorial multi-unit reverse auction where services have unit demand and potentially different participation costs.

Unsatisfactory Alternate Approach. An alternate approach is where the servers first commit and open their asking prices, and the reward distribution function of the client determines the rewards for each servers. With this knowledge, the servers perform the repeated squaring and post their solutions. This way, the servers can invest in the computation depending on the reward price that they are guaranteed by the function $\mathcal{R}$. As only a few servers are selected here, unlike in OpenSquare, servers are guaranteed reward for their computation.

However, there is a potential DoS attack: the servers may, after the reward price guarantee, fail to post solutions. We can require the servers which are guaranteed a non-zero reward price to lock a collateral V_s to the contract along with their asking price commitment to prevent this attack; they shall get back the collateral amount only if they post a valid solution, and otherwise, the collateral V_s is transferred to the client.

Firstly, the alternative approach cannot address the inherent disparity between cost of computation to the servers and value of the computation to the client. Assume that the client will lose L

coins in value in his extraneous application if his squaring computation request does not get solved within t_{sol}, and that required computation cost is $\ll L$. This forces the client to set the solution collateral as $V_s > L$, so that in case the chosen server fails to post a valid solution, the client is not at a loss. This leads to two problems: (a) the client by setting V_s publicly reveals information about the request's value L which is a privacy concern, and (b) if L is very large (depending on the client's application), many servers may simply not possess enough collateral $V_s > L$ to participate in the auction. This can centralize the outsourcing market with only wealthy servers remaining, which we would like to avoid due the selective censoring possible.

Secondly, a server that crashes and is not able to post a solution within the solving phase, is bound to lose its collateral V_s and the cost of computation it has performed so far. In contrast, in OpenSquare, a crashing server only loses the cost of computation performed so far. Nevertheless, our cryptographic protocol as well as our smart contract can be easily adopted to this alternative. We discuss some game-theoretic aspects of this approach in Section 4.3.

1.3 Related Work

Time-lock puzzles were first proposed by Rivest, Shamir and Wagner [36] that used the sequentiality of repeated modular squaring in the RSA group as the fundamental building block for realizing the timed primitive. The same building block was used along with techniques to introduce homomorphism, in the time-lock puzzle constructions of Malavolta and Thyagarajan [27]. The above two constructions of time-lock puzzles enjoy being implemented in practice and leading to several applications [9, 17, 26, 40, 41]. Practical constructions of verifiable delay functions [8, 44, 35] use the same building block as the TLPs above. VDFs have found themselves several applications [31, 12, 39] in the context of blockchain consensus [15, 31].

Theoretical lower bounds have been studied on the adversarial speed up for the repeated modular squaring operation [45]. In terms of practical studies, Thyagarajan et al. [40] studied the varying times required by different AWS machines with different computing powers, for computing the same number of squaring operations. With the advent of VDFs, there has been tremendous amount of development in building specialized hardware like ASICs that achieve high efficiency for repeated modular squaring [1, 30, 2]. There have also been VDF competitions [42], where winners are rewarded for achieving non-trivial speed up in computing a said number of sequential squarings.

Standard auction design often aims to maximize seller revenue or lower buyer cost (optimal mechanism design) [34], and/or maximize social welfare or minimize social cost. In our setting, for the decentralized protocol to prosper, the system explicitly asks for liveness guarantees from a mechanism. Although market thickness - increasing the amount of bidders - outweighs the benefit from finding an optimal reserve price for less bidders [10], an auction usually does not concern itself with attracting bidders. The scene complicates when we consider entry costs since services perform computations, pay gas, etc before they submit bids. In an auction with entry cost, an auctioneer can potentially exert admission fees and/or entry cap [32], hold preliminary auctions [47, 6], organize sequential entry [29, 11, 37] to limit the number of entrants and increase seller revenue. But the smart contract auctioneer in our design does not have active coordination and our goal is to accommodate more bidders (if feasible) instead of limiting the number of entrants to wholeheartedly minimize buyer cost. So these designs are not applicable. Besides, each service only submits the bid once so ascending/English or iterative auctions including Walrasian auction do not apply.

Essentially we desire a reverse auction design considering entry costs that induces more entrants and keeps the price low for users. Samuelson [38] discusses inducing optimal entry in a procurement auction with entry setting and picks one winner. Maskin [28] presents the optimal selling procedure for unit demand buyers in multi-unit standard auction without entry costs. We follow the philosophy in these two works to construct a multi-unit reverse auction with entry.

2 PRELIMINARIES

We introduce the formal notions and briefly recall the relevant cryptographic background. The security parameter is denoted by $\lambda \in \mathbb{N}$ and $x \leftarrow \mathcal{A}(\mathsf{in}; r)$ is the output of the algorithm $\mathcal{A}$ on input in using $r \leftarrow \{0,1\}^*$ as its randomness. We often omit this randomness and only mention it explicitly when required. We consider *probabilistic polynomial time* (PPT) machines as efficient algorithms and use *parallel random access machines* (PRAM) to model machines with parallel processing power.

Verifiable Delay Functions. The primitive was proposed by Boneh et al. [8], that allows to enforce a delay on some computation. It has teh following interfaces. The setup algorithm Setup outputs the public parameters with respect to the security parameter 1^λ and a time parameter $\mathbf{T}$. The public parameters encode the function domain $\mathcal{X}$ and the range $\mathcal{Y}$. The instance generation algorithm Gen outputs a random instance x from the domain $\mathcal{X}$. The evaluation algorithm Eval outputs a delayed output $y \in \mathcal{Y}$ and a proof of correct computation π, for a given instance x. We finally have the verify algorithm Verify that verifies if the output y and the proof π are valid with respect to an instance x or not.

In terms of efficiency, we require that the setup and the instance generation algorithms run in time $p(\lambda)$ for some polynomial p, whereas the running time of the verification algorithm must be bounded by $p(\log(\mathbf{T}), \lambda)$. For the evaluation algorithm, we require it to run in parallel time exactly $\mathbf{T}$. We require the notion of sequentiality and soundness. Sequentiality intuitive says that a PRAM adversary cannot compute the correct delayed output y for some random instance x, significantly before time $\mathbf{T}$. The soundness property guarantees that the adversary cannot convince a honest verifier to output 1 given a proof and an incorrect delayed output $y' \neq y$. The formal definitions are deferred to Appendix A.

Wesolowski [44]'s VDF construction satisfies the above notions. His work also considers the notion of watermarked VDF which will be useful in our work. Intuitively, watermarking a VDF evaluation with an identifier (or a public key) pk means that another user cannot claim the VDF evaluation as his own. In other words, if an evaluator watermarks his evaluation (y, π) with his public key pk, another user with identifier $pk' \neq pk$ cannot compute (y, π') faster than $\mathbf{T}$. To facilitate this we extend the syntax by letting Eval and

Setup$(1^\lambda, \mathbf{T})$: Sample λ bit primes p and q. Set $N := pq$. Output $pp := (\mathbf{T}, N)$.
Gen(pp): Sample $x \leftarrow \mathbb{Z}_N^*$ and output x.
Eval(pp, x, pk): Parse $pp := (\mathbf{T}, N)$. Set $x_2 := H(x, pk)$. Compute $y_1 := x^{2^\mathbf{T}} \bmod N$ and $y_2 := (x_2)^{2^\mathbf{T}} \bmod N$. Generate $\ell := H_p(x, y_1, x_2, y_2)$ and compute q and r such that $2^\mathbf{T} = q\ell + r$. Set $\pi := (\ell, x^q, (x_2)^q)$ and output $((y_1, y_2), \pi)$.
Verify(pp, x, y, π, pk): Parse $y := (y_1, y_2)$ and $\pi := (\ell, Q_1, Q_2)$. Compute $x_2 = H(x, pk)$. Check if $\ell \stackrel{?}{=} H_p(x, y_1, x_2, y_2)$, and output 0 otherwise. Check if $y_1 \stackrel{?}{=} (Q_1)^\ell x^r$ and $y_2 \stackrel{?}{=} (Q_2)^\ell (x_2)^r$, where $r := 2^\mathbf{T} \mod \ell$. If any of the check fails, output 0.

Figure 2: Modification of the watermarked VDF construction from [44] that is used in OpenSquare.

Verify additionally take a public key pk as input. For the sake of completeness, we describe the watermarked VDF construction Figure 2 that we use in this work. We consider two hash functions $H_p : \{0,1\}^* \to \mathsf{PRIMES}_{2\lambda}$ and $H : \{0,1\}^* \to \mathbb{Z}_N^*$, where $\mathsf{PRIMES}_{2\lambda}$ consists of all 2λ bit primes. The VDF proofs can also be aggregated by the prover for faster verification as described in [44].

Ethereum Smart Contracts. Ethereum [46] is a decentralized virtual machine (Ethereum Virtual Machine or EVM) that runs *smart contracts*. Smart contracts facilitate complex conditional payments between users apart from standard coin transfers. A smart contract is a program written in EVM bytecode. It can transfer money to users based on its programming. It is triggered using transactions which contain information about the function to call and input data. The miners include these transactions in the blocks and update the global state. The miners receive transaction fees as incentive to process execute transactions. In Ethereum, the term *gas* is used to denote a unit of computation. The fees is paid via *gas price* which is the amount of ETH per gas a user is willing to pay. The miners are incentivized to process higher gas priced transactions[1]. Every instruction inside the contract has a pre-defined amount of gas as its corresponding fee. This is specified in the Ethereum reference [46]. The caller of a contract has to pay the amount of gas that is enough to perform all the steps of the contract call. Every call starts with a minimum gas cost of 21, 000 to setup the EVM, and for every instruction such as loading the arguments, moving values, etc. are charged with a fixed gas cost per operation. For instance, a Keccak256 opcode charges 30 gas to hash an empty string, 36 gas to hash any amount of data up to 32 bytes, and 42 gas for data of size $33 - 64$ bytes, and so on. The fee to the miner is paid in Ether[1].

Mechanism Design. Game theory studies interactions between competing, strategic and rational agents, and one aims to solve for equilibrium outcomes according to solution concepts. Mechanism design (MD) fixes a set of desrired outcomes, and one can design mechanisms for agents to interact with each other that yield these outcomes. The insight of MD is to account for both resource and incentive constraints [33]. One special type of mechanism is direct-revelation mechanisms where there exists a hypothetical mediator that collects private information from agents and recommends actions to them. If the mechanism encourages compiliant/honest behaviors from agents, then it is called incentive compatible (IC).

Reverse and forward auction. One basic mechanism is auction [25]. Two typical objective functions for auctions are to maximize social welfare (minimize social cost) and to maximize seller revenue (minimize buyer cost). A procurement/reverse auction has a single buyer and multiple suppliers, who place bids to earn the opportunity to provide the buyer an item at a price. In a standard/forward auction, a seller seeks to sell item(s) to potential buyers. A reverse auction with a ceiling price is mathematically equivalent to a standard auction. So the discussions in both contexts can translate.

Unit demand bidders A unit demand bidder desires only a single item, which can be one of multiple homogeneous or heterogeneous goods. For k identical items with unit demand bidders (what we consider in this work), k-Vickrey auction achieves dominant-strategy incentive compatibility and social welfare maximization. Note that an auction is *dominant-strategy incentive compatible (DSIC)* if for all bidders, truth-telling (weakly) dominates other strategies. k-Vickrey auction has allocation rule that allocates the items to k highest bidders and payment rule that the winners pay the $(k+1)$-th highest bid. Also note that to procure k identical items, one can construct a reverse version of k-Vickrey auction and set a ceiling price.

3 DECENTRALIZED REPEATED MODULAR SQUARING SERVICE

In this section, we present our OpenSquare protocol, where a client can use a smart contract to post a request, and the services are required to compute the requested number of repeated modular squarings as the solution, and get paid. The client can be a user wanting to solve a time-lock puzzle [36] or compute a delay function [8, 35, 44], and outsources the necessary computation through our decentralized solving service protocol. We model the core functionality of OpenSquare in the form an ideal functionality $\mathcal{F}_{\mathsf{SolS}}$ in the UC framework [13], which can be found in Appendix B.

For simplicity, we consider a single client and a set of services to constitute the entire user set in the system. The users interact with a smart contract C_{OpSq} on the chain by posting transactions. The client has $(g, \mathbf{T}, N)$ (where N is a RSA modulus) that he obtains from the application he is participating in, that requires him to solve a time-lock puzzle or evaluate a VDF. The problem ϕ that the client wants solved is to compute $g^{2^\mathbf{T}} \mod N$. The client can outsource this computation through OpenSquare and obtain the solution to ϕ.

3.1 OpenSquare Smart Contract

We now describe the smart contract C_{OpSq}, which is posted on the chain and facilitates the above operations. On a high level, the contract offers the following interfaces: (1) request registration function `NewRequest()` to receive a request and reward amount from the client, (2) a solution submission function `SubmitSolution()` to

[1]This is an over-simplification. Actually, a user pays x = gas limit× gas price. A miner executes the transaction and observes that the transaction costs x'. The miners make a profit out of $x - x'$. So it is not technically correct that the miners just take a look at the gas price. Actually they maintain a priority queue in their mempool and reap the most profitable transactions while mining every block.

receive solutions and service collaterals from services, (3) an asking price function Ask() to let services reveal their asking prices, (4) a reward claim function Claim() to let services claim their reward amounts and their locked collateral, (5) a complaint function Complaint() to receive complaints from any user about a solution along with the complaint fee and finally (6) the refund function Refund() that transfers leftover reward coins from the contract to the client. The contract is described in detail in Figure 3. We require a hash function $H' : \{0,1\}^* \to \{0,1\}^\lambda$.

3.2 OpenSquare Protocol

The client having $(g, \mathbf{T}, N)$ posts a request that offers a reward up to p coins with a reward distribution function $\mathcal{R}$. He also determines the duration of the different phases, particularly the solving phase, asking phase, complaint phase, and the refund time. If some service has posted a solution (y, π) to the contract, the client verifies if the solution is valid with respect to the service's public key using the algorithm Verify(). If it is valid, the client has obtained $y = (y_1, y_2)$ with $y_1 = g^{2^{\mathbf{T}}}$ and does not need to act further on the solution. If at least one of the services posts a valid solution, the client can use it for its application (time-lock puzzle or VDF). Valid solutions recorded after this point are of no use to the client except to incentivize services to participate in the service protocol Section 4. During the refund phase, the client recovers any coins left in the contract that is left over after rewarding the services for their solution.

To post a solution, the service runs the evaluation algorithm Eval() of the watermarked VDF Figure 2 to compute (y, π) where $y_1 := g^{2^{\mathbf{T}}}$ and $y_2 := x^{2^{\mathbf{T}}}$ with $x := H(g, pk)$. Here pk is the public key of the service, to which the service obtains any reward from the contract. The service also sends a commitment h to its asking price as part of the solution. It later reveals the asking price a and randomness r such that $h = H'(a, r)$ during the asking phase. The service claims its collateral and any reward determined by the reward distribution function from the contract before the request is inactivated.

Any party can complain against a candidate solution from a service. The complaint has to be made within t_{cmpl} amount of time after the solution is posted to the contract. The complaint is valid if Verify() algorithm returns 0 on the particular solution. A detailed description of our protocol is given in Figure 4.

Security. Our protocol with the aid of the contract $\mathcal{C}_{\mathsf{OpSq}}$ securely realizes the ideal functionality $\mathcal{F}_{\mathsf{SolS}}$ when ϕ is that of the repeated modular squaring problem. The security intuitively follows from the soundness of the watermarked VDF, which ensures that only valid solutions are rewarded according to the reward distribution function $\mathcal{R}$ chosen by the client. Invalid solutions can be complained against, and no reward is possible. The formal theorem statement and the security analysis are deferred to Appendix C.

3.3 Discussion

In this section, we discuss several important properties and extensions for our protocol.

Unlinkable Outsourcing of Puzzle Solving. As noted earlier, the client can use our protocol to outsource solving of time-lock puzzles [36, 27]. Time-lock puzzle constructions from [36, 27] have their puzzles Z of the form $Z := (Z_1, Z_2)$. To solve the time-lock puzzle, the user has to compute $(Z_1)^{2^{\mathbf{T}}} \mod N$ and using this value, he can retrieve the embedded message from Z_2. In our protocol, the client can outsource the computation of $(Z_1)^{2^{\mathbf{T}}} \mod N$ by setting the request as $(Z_1, \mathbf{T}, N)$. However, if the puzzle Z is part of some application and is known to the users in the system, it is possible that the services can launch a *denial of solving* attack on the client. That is, the services may not want the client to know the embedded message inside the puzzle in time, and therefore not solve the request. This attack works because the request $(Z_1, \mathbf{T}, N)$ is linkable to the puzzle Z.

We can prevent this attack, by making the request unlinkable to the corresponding time-lock puzzle. The client can do this by re-randomizing the request, set $Z_1' := Z_1 \cdot g^r$ for some random r chosen uniformly over the randomness domain and generator g. The request is now set as $(Z_1', \mathbf{T}, N)$ and since r is chosen uniformly at random, Z_1' looks completely random and unlinkable to Z_1 in the eyes of the services. The services then compute $(Z_1')^{2^{\mathbf{T}}} \mod N$ as part of their solution.

What is remaining to show is how the client can retrieve $Z_1^{2^{\mathbf{T}}}$ from $(Z_1')^{2^{\mathbf{T}}}$ efficiently, without having to compute $Z_1^{2^{\mathbf{T}}}$ from scratch. Fortunately, this is possible with the RSW-based homomorphic time-lock puzzle constructions from [27]. Their constructions have a setup algorithm run at the beginning of the system that returns $(g, g^{2^{\mathbf{T}}}, \mathbf{T}, N)$ to users as part of the public parameters, where g is the generator of $\mathbb{J}_N^*$ (subgroup of $\mathbb{Z}_N^*$ with Jacobi symbol +1). Now given $(Z_1')^{2^{\mathbf{T}}}$ and the public parameters $(g, g^{2^{\mathbf{T}}}, \mathbf{T}, N)$ the client with knowledge of r, can first compute $\left(g^{2^{\mathbf{T}}}\right)^r$ and retrieve

$$s := \frac{(Z_1')^{2^{\mathbf{T}}}}{\left(g^{2^{\mathbf{T}}}\right)^r} := \frac{(Z_1 \cdot g^r)^{2^{\mathbf{T}}}}{\left(g^{2^{\mathbf{T}} \cdot r}\right)} := \frac{(Z_1)^{2^{\mathbf{T}}} \cdot g^{2^{\mathbf{T}} \cdot r}}{\left(g^{2^{\mathbf{T}} \cdot r}\right)} := (Z_1)^{2^{\mathbf{T}}}$$

Reward Distribution. Notice that the reward distribution function $\mathcal{R}$ is a function of the asking prices. Also the client rewards k valid solutions with possibly differing reward amounts and this is checked in step e. of the Claim() function of the contract. This is to incentivise a large pool of services to participate and post valid solutions. We discuss this in more detail in Section 4.

Non-transferrability of Solutions. Services post their solutions in plain and only commit to their asking price which they later reveal during the asking phase. This could lead to services copying solutions of other services as their own. Using the watermarked VDF (Figure 2) evaluation, we simultaneously ensure the following: (1) the client learns $g^{2^{\mathbf{T}}}$ the solution he is looking for, and (2) no service can produce a valid solution by copying another solution in significantly better time than $\mathbf{T}$. The latter guarantee follows from the watermarking and the sequentiality of the squaring operation (Section 2).

Sybil Resistance. During solving, services have to evaluate $y_1 := g^{2^{\mathbf{T}}}$ and $y_2 := x^{2^{\mathbf{T}}}$, where $x := H(g, pk)$ (as in Figure 2). Notice that y_2 is specific to the service's public key pk and since the hash function H is modelled as a random oracle (and is therefore collision resistant), with overwhelming probability y_2 is not valid with

The contract C_{OpSq} maintains hash maps $\mathcal{M}_{Act}, \mathcal{M}_{Rsp}, \mathcal{M}_{ask}$ that are initialised to empty maps during the contract creation. The contract implements the following functionalities:

- `NewRequest()`:
  a. A transaction tx_{req} from pk_C calls the function with inputs $(\mathsf{Rq}, t_{sol}, t_{Rfnd}, t_{cmpl}, t_{ask}, k, R, V_s, V_c)$
  b. Request id is generated as $\mathtt{RqID} := H'(\mathsf{Rq})$
  c. Parse reward $R := (p, \mathcal{R})$, where p is the reward budget and $\mathcal{R}$ is the reward distribution function.
  d. Ensure the transaction tx_{req} sends p coins to the contract, where the address is denoted by $\mathsf{addr}_{\mathtt{RqID}}$
  e. Ensure $t_{Rfnd} > t_{ask} > t_{sol}$ and $t_{cmpl} < t_{Rfnd} - t_{sol}$
  f. Record the current block number as B_{strt} and set $\mathsf{ctr}_{\mathtt{RqID}} := 0$
  g. Add $(R, V_s, V_c, t_{sol}, t_{cmpl}, t_{Rfnd}, t_{ask}, k, pk_C, B_{\mathsf{strt}})$ to active request map $\mathcal{M}_{Act}$ using the request id `RqID` as key, and return `RqID`
- `SubmitSolution()`:
  a. A transaction tx_{resp} from pk_S calls the function with inputs $(\mathtt{RqID}, y, \pi, h)$, in the block B_{num}
  b. Retrieve $(\cdot, V_s, \cdot, t_{sol}, \cdot, \cdot, \cdot, \cdot, \cdot, B_{\mathsf{strt}}) \leftarrow \mathcal{M}_{Act}$ using `RqID` as the key. If no such entry exists in the list, then do nothing and return 0
  c. Generate response id $\mathtt{RspID} := H'(y, \pi, pk_S, h)$
  d. Ensure $B_{num} < B_{\mathsf{strt}} + t_{sol}$ and transaction tx_{resp} locks V_s amount to the contract, at the address denoted by $\mathsf{addr}_{\mathtt{RspID}}$
  e. Store $(\mathtt{RqID}, h, B_{num})$ in the response map $\mathcal{M}_{Rsp}$ using the key `RspID`
- `Ask()`:
  a. A transaction tx_{ask} from pk_S calls the function with inputs $(\mathtt{RqID}, \mathtt{RspID}, a, r)$. Let B'_{num} be the current block number
  b. Retrieve $(\cdot, \cdot, \cdot, t_{sol}, \cdot, \cdot, t_{ask}, \cdot, \cdot, B_{\mathsf{strt}}) \leftarrow \mathcal{M}_{Act}$ using `RqID` as the key. If no such entry exists in the list, then do nothing and return 0
  c. Ensure the $B_{\mathsf{strt}} + t_{sol} < B'_{num} < B_{\mathsf{strt}} + t_{ask}$
  d. Retrieve $(\mathtt{RqID}, h, B_{num}) \leftarrow \mathcal{M}_{Rsp}$ using the key `RspID`
  e. If $h \neq H'(a, r)$, then return 0. Else store a in $\mathcal{M}_{ask}$ with keys `RqID` and `RspID` and return 1
- `Claim()`:
  a. A transaction tx_{claim} from pk_S calls the function with inputs $(\mathtt{RqID}, y, \pi, pk_S, h)$. Let B'_{num} be the current block number
  b. Retrieve $(R, V_s, \cdot, \cdot, t_{cmpl}, t_{Rfnd}, t_{ask}, k, \cdot, B_{\mathsf{strt}}) \leftarrow \mathcal{M}_{Act}$ using the key `RqID`. If no such entry exists, then return 0
  c. Compute $\mathtt{RspID} := H'(y, \pi, pk_S, h)$
  d. Retrieve $(\mathtt{RqID}', h, B_{num}) \leftarrow \mathcal{M}_{Rsp}$ using the key `RspID`, and if no such entry exists or $\mathtt{RqID}' \neq \mathtt{RqID}$, do nothing and return 0
  e. Check if $\mathsf{ctr}_{\mathtt{RqID}} = k$, if so, transfer V_s coins to pk_S and remove the entry $(\mathtt{RqID}', h, B_{num})$ with key `RspID` from $\mathcal{M}_{Rsp}$
  f. If we have $\mathsf{ctr}_{\mathtt{RqID}} = j < k$, ensure $B'_{num} > B_{num} + t_{cmpl}$ and $B'_{num} > B_{\mathsf{strt}} + t_{ask}$.
  g. Retrieve a list A consisting of all a in $\mathcal{M}_{ask}$ with key `RqID`, and a^* in $\mathcal{M}_{ask}$ with key `RspID`
  h. Parse $R := (\cdot, \mathcal{R})$ and retrieve the remaining reward p for the request from the contract address $\mathsf{addr}_{\mathtt{RqID}}$. Compute $\mathcal{R}(A, a^*, j) = p^*$. If $p^* \leq p$, transfer p^* coins from $\mathsf{addr}_{\mathtt{RqID}}$ and V_s coins from $\mathsf{addr}_{\mathtt{RspID}}$ to pk_S.
  i. Set $\mathsf{ctr}_{\mathtt{RqID}} = j + 1$ and remove the entry $(\mathtt{RqID}', h, B_{num})$ with key `RspID` from $\mathcal{M}_{Rsp}$ and return 1.
- `Complaint()`:
  a. A transaction tx_{cmpl} from pk_A calls the function with inputs $(\mathsf{Rq}, y, \pi, h, pk_S)$. Let B'_{num} be the current block number
  b. Compute $\mathtt{RqID} := H'(\mathsf{Rq})$ and retrieve $(R, V_s, V_c, t_{sol}, t_{cmpl}, t_{Rfnd}, t_{ask}, k, pk_C, B_{\mathsf{strt}}) \leftarrow \mathcal{M}_{Act}$ using the key `RqID`. If no such entry exists in the list, then do nothing and return 0.
  c. Compute $\mathtt{RspID} := H'(y, \pi, pk_S, h)$ and retrieve $(\mathtt{RqID}', h, B_{num}) \leftarrow \mathcal{M}_{Rsp}$ using the key `RspID`. If no such entry exists in $\mathcal{M}_{Rsp}$ or if $\mathtt{RqID} \neq \mathtt{RqID}'$, do nothing and return 0
  d. Ensure $B'_{num} < B_{num} + t_{cmpl}$
  e. Ensure the transaction tx_{cmpl} locks V_c amount of coins to the contract
  f. Parse $\mathsf{Rq} := (\mathsf{req}, \mathbf{T})$
  g. Check whether $\mathsf{Verify}((\mathbf{T}, N), y, \pi, pk_S) = 0$, if so transfer V_s coins from $\mathsf{addr}_{\mathtt{RspID}}$ to pk_A. Remove $(\mathtt{RqID}, h, B_{num})$ from the list $\mathcal{M}_{Rsp}$ with key `RspID`, and return 1
  h. If the above check fails, return 0
- `Refund()`:
  a. A transaction $tx_{\mathtt{Refund}}$ from pk_C calls the function with inputs $(\mathtt{RqID})$. Let B'_{num} be the current block number
  b. Retrieve $(R, V_s, V_c, t_{sol}, t_{cmpl}, t_{Rfnd}, t_{ask}, k, pk_C, B_{\mathsf{strt}}) \leftarrow \mathcal{M}_{Act}$ using the key `RqID`.
  c. Ensure $B'_{num} > B_{\mathsf{strt}} + t_{Rfnd}$
  d. Parse $R := (\cdot, \mathcal{R})$
  e. Transfer the remaining p coins (corresponding to `RqID`) from the contract address $\mathsf{addr}_{\mathtt{RqID}}$ to pk_C, and remove the entries from the active list $\mathcal{M}_{Act}$, $\mathcal{M}_{Rsp}$ and $\mathcal{M}_{ask}$ for the request id `RqID`, and return 1

Figure 3: Description of the main functions of contract C_{OpSq}

Client routine

The client has public parameters $(\mathbf{T}, N)$ and has an element $g \in \mathbb{Z}_N^*$ that he wants to be evaluated. The client does the following steps:

- Set a request $\mathsf{req} := (g, \mathbf{T}, N)$ and choose $t_{sol}, t_{Rfnd}, t_{cmpl}, t_{ask}$. Assign a reward procedure $R := (p, \mathcal{R})$. Chooses V_s and V_c as the coins to lock during registering solution and making a complaint, respectively.
- Post a transaction tx_{req} on the chain that calls C_{OpSq}.`NewRequest`() with inputs $(\mathsf{Rq}, t_{sol}, t_{Rfnd}, t_{cmpl}, t_{ask}, k, R, V_s, V_c)$ along with sending p coins to the contract C_{OpSq}'s address.
- Store the id of his request `RqID`.
- If there is a call to C_{OpSq}.`SubmitSolution`() in some transaction with public key pk_S, with inputs $(\mathtt{RqID}, y, \pi, h)$, such that $\mathsf{Verify}((\mathbf{T}, N), g, y, \pi, pk_S) = 1$, then retrieve y as the required solution.
- After t_{Rfnd} number of blocks have passed since posting the request req, post a transaction $tx_{\mathtt{Refund}}$ that calls C_{OpSq}.`Refund`() with input `RqID`, on the chain.

Service routine

The service observes a request id `RqID` in the contract C_{OpSq}. It does the following steps:

- Retrieve the transaction tx_{req} that called the contract C_{OpSq}, with inputs $(\mathsf{Rq}, t_{sol}, t_{Rfnd}, t_{cmpl}, t_{ask}, R, V_s, V_c)$. Here we have $R := (p, \mathcal{R})$ and $\mathsf{Rq} := (g, \mathbf{T}, N)$.
- Runs the VDF evaluation $(y, \pi) \leftarrow \mathsf{Eval}((\mathbf{T}, N), g, pk_S)$, where pk_S is the service's key.
- Choose a asking price a that is smaller than p and commit to it by setting $h := H'(a, r)$ where $r \leftarrow \{0,1\}^*$.
- Post a transaction tx_{resp} on the chain which calls C_{OpSq}.`SubmitSolution`() with input $(\mathtt{RqID}, y, \pi, h)$. The transaction additionally locks V_s coins to the contract C_{OpSq}. Store the response id `RspID` generated by the contract C_{OpSq}.
- Post a transaction tx_{ask} on the chain that calls C_{OpSq}.`Ask`() with inputs $(\mathtt{RqID}, \mathtt{RspID}, a, r)$.
- To claim the reward (and the locked V_s coins), post a transaction tx_{claim} on the chain that calls C_{OpSq}.`Claim`() with inputs $(\mathtt{RqID}, y, \pi, pk_S, h)$.

Complaint routine

Any user on the networks can complain against a response with id `RspID` for the request `RqID`. The user does the following:

- Retrieve the transaction tx_{resp} that recorded the response `RspID` with inputs $(\mathtt{RqID}, y, \pi, h)$ and public key pk_S. Retrieve the request $\mathsf{req} := (g, \mathbf{T}, N)$ corresponding to `RqID`.
- Check if $\mathsf{Verify}((\mathbf{T}, N), g, y, \pi, pk_S) = 0$. If so, post a transaction tx_{cmpl} on the chain that calls C_{OpSq}.`Complaint`() with inputs $(\mathsf{Rq}, y, \pi, h, pk_S)$. The transaction additionally locks V_c coins to the contract.

Figure 4: OpenSquare routines for Clients, Servers and any user in the blockchain network.

respect to a public key pk' where $pk' \neq pk$. This means that the service would have to compute a fresh y_2' for a different public key pk', which by the sequentiality property of the VDF ensures that the service has to spend time $\mathbf{T}$ in computing y_2'. Therefore we ensure that solutions are only rewarded if $\mathbf{T}$ computational steps are performed in computing it and services cannot simply duplicate their solutions with different public keys and get rewarded.

Optimistic Low On-chain Computational Cost. In the optimistic scenario where the client and the services are honest, no expensive cryptographic operation needs to be performed by the contract. Since all users are honest, no wrong solutions are registered at the contract, and no (rational) user has to call the `Complaint`() function of the contract. Only operations the contract performs are hashing operations, storing and retrieving elements from hash maps, and transferring of coins between the contract address and a user address.

Solution Collateral and Complain Fee. Cryptographic operations (verification algorithm Verify() of the watermarked VDF) are run in the `Complaint`() function. Miners who run the contract perform the computation and expect to be reimbursed for the computational cost, usually paid from the user who calls the function of the contract. In our protocol, the complainant calls the `Complaint`() function of the contract, and the cost of computing the `Complaint`() function is compensated from the complaint fee V_c. Therefore it is important that the complain fee V_c is sufficient enough to run the `Complaint`() function of the contract for the request. To incentivize complainants to complain against wrong solutions, we set the solution collateral $V_s > V_c$. In the event of a successful complaint, the complainant loses V_c coins but gains the service's collateral V_s coins, which is more than what he lost, thus encouraging users to complain against wrong solutions.

4 OPENSQUARE MECHANISM DESIGN

In this section, we define the reward distribution function in OpenSquare through a (multi-unit) reverse auction with entry costs such that the protocol achieves the following objectives:

- **Participation/Liveness**: each request req feasible for the system is solved within the solve phase t_{sol} within posted reward budget p with high probability. The optimal solving probability is 1. By feasible, we mean that there exists at least a service that can accomplish the computations in time t_{sol} and post a valid response with an asking price no greater than the budget.
- **Minimized buyer cost**: the amount of reward distributed for valid solutions is minimized to ensure the clients pay less.

The first goal is concerned with offering incentives for providing the service and the second goal is to enhance the affordability of the service. For the liveness goal, instead of only considering maintaining "enough" services in the market so that a constant number of requests can be attended at a time, we aim to keep as many services in the market as possible. One motivation is to prepare for high throughput in case of a potential outburst of request volumes. Another important reason is to accommodate more services to protect users against DDoS attacks.

Modelling. We model OpenSquare protocol as a procurement auction with entry costs, and the number of participants is not fixed. Potential bidders (services) know who might be in the same auction (request) but only know the entrants after they bid. In the decentralized setting, we assume the auctioneer (contract C_{OpSq}) can neither cap the number of entrants directly nor coordinate potential bidders actively, e.g matching services to requests.

We assume standard risk neutral[2] agents with quasi-linear utilities[3]. Note that here risk-neutral assumption can also be relaxed to risk averse [21], which reduces procurement costs. Agents have a unit supply for requests so that one bidder only participates in one auction at a time. This is because of the sequential nature of the repeated modular squaring and that services cannot copy solutions (non-transferability and Sybil resistance).

We focus on direct mechanisms where participants reveal their private values to the mechanism. The nature of OpenSquare imposes endogenous and potentially heterogeneous entry costs (computations, amortized setup costs, etc.) before bidding. The entry costs are deadweight loss, which is not compensated directly and not received by any party. These costs may not be homogeneous because we assume non-identical services which potentially scatter geographically and differ in computational capacity.

The client and other services have complete information over a service's types but not its private values. Further we assume services have independent identical cost distribution $(F, f, \underline{c}, \overline{c})$, where F is the cumulative distribution function (CDF), f is the probability density function (PDF), $\underline{c}$ and $\overline{c}$ are the supports. Their actual cost values are independent private values (IPV) drawn from F. We adopt the regularity assumption on the cost distribution, meaning that for a cost c_i the virtual value function, $\frac{F(c_i)}{f(c_i)}$ is monotone non-decreasing in c_i. This means that F is log-concave function, eg., uniform, normal and exponential distribution.

4.1 Single Request Auction with Entry

Let C be a client who wants her request $(g, \mathbf{T}, N)$ solved. Client C can assess the historic request complexity, the corresponding solving times, and payments to decide whether its request is feasible for this system. The auctioneer picks k winners ($k \geq 1$) from a single-request auction. C is satisfied as long as she obtains the solution and pays as little as possible. We first represent the service in an auction fashion.

A single buyer C wants to obtain at least one indivisible solution to her request req from n potential services in time t_{sol} through a sealed-bid reverse auction. C sets a ceiling price p. This gives $r = p/k$ as the individual ceiling price for winners. Auctioneer selects and pays k winner(s) on behalf of C. Services capable of performing such computations within ceiling price r consider participating. To make an entry decision, each service S_i draws a projected cost c_i from F. After performing repeated squarings, S_i realizes cost a_i for providing the solution and places bid $b_i = b(a_i)$, where $b(\cdot)$ is a monotonically increasing function in a_i.

We denote the set of services $\mathcal{N} := \{S_1, \ldots, S_n\}$. We define the auction as a vector of two functions $A = (\mathbf{x}, \mathbf{p})$ where $\mathbf{x}$ is the allocation/demand function and $\mathbf{p}$ is the payment function. The reward distribution function $\mathcal{R}$ comprises of configuration of p (reward budget), k (number of winners) and the auction $(\mathbf{x}, \mathbf{p})$. In the first stage of the game, services simultaneously determine entry and in the second round, if at least one service participates, we run a potentially multi-unit sealed-bid auction with a ceiling price r.

4.1.1 Model entry. The entry decision is made *after* potential bidders learn their types. Services can approximate the computation costs for a given request. They make entry decisions *simultaneously* because, in our setting, there's no meaningful identity bound with services or coordinator to guide sequential entries. Even the auctioneer only knows which services are in the auction after receiving valid bids from them.

Solution concept. In the first stage of the game where services decide entry, similar to [38], we solve for the equilibrium where services with some break-even cost c^* are indifferent towards entry (we determine c^* as we proceed). If a service has costs above c^*, it does not participate. Our focus for the entry game is to determine the number of bidders we can accommodate by picking k winners.

Order of events. (1) Auctioneer has access to F and designs the auction; (2) services learn their types by drawing a projected cost from F; (3) services decide whether to enter into the auction; (4) auction participants submit bids along with solutions; (4) auctioneer decides and realizes payments after the auction closes.

In an auction A with individual ceiling price r, a service S_i with borderline projected cost c^* has winning probability

$$(1 - F(c^*))^{n-k}$$

Adding more agents decreases this probability. Server S_i's optimal bid is r and its expected profit from participating in the auction is

$$\pi(c^*) = (r - c^*)(1 - F(c^*))^{n-k}$$

When $\pi(c^*) = c_p$, the bid-preparation cost, S_i is indifferent towards entry and this is the equilibrium entry condition. Here c_p is known, can be related to computation costs, and is the same for all potential bidders with break-even projected cost c^*.

To have a "desired" c^*, we start with minimizing buyer cost. We calculate buyer cost C_b as follows.

$$\begin{aligned} C_b =& c_0(1 - F(c^*))^n + \int_{\underline{c}}^{c^*} k c f_{(k)}(c) dc \\ &+ n \int_{\underline{c}}^{c^*} F(c)(1 - F(c))^{n-k} dc + n c_p F(c^*) \end{aligned}$$

where $f_{(k)}(c) = n\binom{n-1}{k-1}(1 - F(c))^{n-k} F(c)^{k-1} f(c)$ is the PDF of the k-th order statistics (with $f_{(1)}$ being the PDF for the minimum), and c_0 is the price the client needs to pay to obtain a solution in an alternative way (outside OpenSquare) or bear the loss of not being

[2]For a risk-neutral agent, receiving an amount $\sum_{i=1}^{n} v_i p_i$ deterministically is the same as receiving each v_i with probability p_i for $i \in \{1, \ldots, n\}$.

[3]Utility for obtaining an item of value v at price p is $(v - p)$.

able to attain a solution. The first term captures the cost of not receiving the solution. The second term is the cost of obtaining k solutions. The third term computes the expected profit for bidders. The last term computes the expected entry costs for all services. We took the first order differentiation of C_b on c^* and arrives at the first-order condition in Equation (1).

$$[c_0(1-F(c^*))^{k-1}-\binom{n-1}{k-1}kc^*F(c^*)^{k-1}-\frac{F(c^*)}{f(c^*)}]\cdot[1-F(c^*)]^{n-k}=c_p \tag{1}$$

After knowing c^*, we can determine r since $\pi(c^*)=c_p$. The aggregate user ceiling price $p=rk$. For a given c^*, the probability of a service having lower cost is $F(c^*)$, then the expected number of entrants is $nF(c^*)$.

4.1.2 Second stage auction. After services making their entry decision, we can start the auction. Suppose $n^* \le n$ services are entrants. Given bids as input, $\mathbf{x}(b_1,\ldots,b_{n^*})$ gives the allocation $(x_1,\ldots,x_{n^*})$ where each x_i is the probability of S_i being the winner in an auction and $\mathbf{p}(b_1,\ldots,b_{n^*})$ gives the payment $(p_1,\ldots,p_{n^*})$. Note that $\mathbf{x}$ gives the ex post winning probability computed by auctioneer after receiving all bids. For the two aforementioned desired goals, we realized participation through interim *Individual Rationality (IR)* in the entry phase. To achieve *minimized buyer cost*, we set the ceiling price r and run a sealed-bid multi-unit auction.

- *The allocation rule* $\mathbf{x}$: is to pick the smallest k bidders below the ceiling price as winners.
- *The payment rule* $\mathbf{p}$: is to pay each winner the minimum of the $(k+1)$-th bid and the ceiling price.

This is dominant-strategy incentive compatible (DSIC) because services have unit supply. We state the following theorem in the IPV setting, which follows from our ceiling price condition. Since we do not change n arbitrarily after derivation, we achieve minimized cost for obtaining k solutions. The proof sketch can be found in Appendix D.

Theorem 4.1. *In single-request auction with n services as potential bidders where* $n \gg 1, \mathcal{R}=(p,k,\mathbf{x},\mathbf{p})$ *as defined induces optimal entry.*

4.2 Multi-Request Auction with Entry

We now consider l requests: $(g_1,\mathbf{T}_1,N_1),\ldots,(g_l,\mathbf{T}_l,N_l)$. Each service has a unit supply and only enters one auction at a time. Let the l requests be feasible for the system. Because cost distributions are public, services are aware of the capacity of all services, meaning that given a request $(g_m,\mathbf{T}_m,N_m), m\in[l]$, S_i knows how many services are capable of solving it (in time within budget). Let $\mathbf{s}=(s_1,\ldots,s_l)$ record the number of services capable of solving the corresponding request and $\mathbf{f^i}=(f_1,\ldots,f_l)$ be the feasibility vector indicating whether S_i is capable of solving the corresponding request. We have a special case: if $\sum f^i=0$, then S_i does not participate and if $\sum f^i \ge 1$, it plays in the first stage entry game.

Difficulties. Unlike in a single-request environment, where a cost-minimizing multi-unit auction suffices, in a multi-request environment, the marketplace needs to dynamically balance between supply and demand. Furthermore, the auctioneer cannot enforce (p,k) configurations on requests. This means that computing VCG[4] payments or minimum weight bipartite matching[5] like price vectors in Walrasian equilibrium cannot be meaningfully implemented, even in static settings. Individual single request auctions may be correlated but are not coordinated. And the ascending auction is not a choice because we prefer a one-time sealed bid auction.

Solution concept. In the multi-request entry game, we continue to first solve for the equilibrium where services with the break-even cost c^* for an auction are indifferent towards entry. But the difficulty is that S_i does not know how many bidders are in each request auction and there is no meaningful way to solve for the maximum expected returns in general without knowing the number of competitors. Since there exist multiple distinct requests, we first internalize the existence of other requests by including opportunity costs for entering into one request auction but not another. We then look at how bidders decide on a specific auction to enter.

Order of events. The order of events in a single auction is similar to before. Now in step (3), services decide which auction to enter.

We organize the auctions according to the number of services capable of solving them (in time within the ceiling price) in ascending order. Without loss of generality, let $\mathbf{s}$ be ascending so that we do not need permutation. $\mathbf{s_i}=(\mathbf{s}\cdot\mathbf{f^i})$ gives the number of potential bidders in auctions feasible for a service S_i. S_i can calculate for each auction A_w in its feasibility set (where $\mathbf{f^i}$ is 1) its opportunity cost as follows: for $\mathbf{f^i}(w)=1$ ($\mathbf{f^i}=1$ at index w), (1) first locate all the other auctions that is not compatible with auction A_w and denote the incompatible set as I_w. By incompatible we mean after solving request $(g_w,\mathbf{T}_w,N_w)$, the other request auction becomes infeasible. (2) For each single-request auction in I_w, we calculate its expected returns with the number of participants at its maximum and we set the opportunity cost c^o_{iw} to be the maximum of these expected returns. If I_w is $\emptyset$, we set $c^o_{iw}=0$ and let $\pi_i(c^*_w)=c_p+c^o_{iw}$ be the break-even entry condition. Following the procedure in single-request environment, a client can solve for c^*_w (Equation (1)) and determine (p,r) to induce desired entry $\mathbf{s}(w)F(c^*_w)$.

Choice of the auction. Bidders have to choose one of the l auctions to participate in. Suppose S_i is faced with two feasible request auctions A_1,A_2 which have n_1,n_2 potential bidders, break-even prices c^*_1,c^*_2 and unit ceiling price r_1,r_2. Let S_i have projected cost $c_1<c^*_1$ for A_1 and $c_2<c^*_2$ for A_2. S_i's expected profits from the two auctions are:

$$\pi_1(c_1)=(r_1-c_1)(1-F(c_1))^{n_1-k_1}$$
$$\pi_2(c_2)=(r_2-c_2)(1-F(c_2))^{n_2-k_2}$$

where $n_1=\mathbf{s}(1)F(c^*_1)$ and $n_2=\mathbf{s}(2)F(c^*_2)$. By comparing the two numbers, S_i can decide which auction to participate in.

We note that in the special case where all requests are feasible for all services, we can simply solve for symmetric equilirbium. For example, let services enter into A_1 with probability p, and A_2 with probability $1-p$. We solve for p in

$$\int_{c_1}^{c^*_1}F(c)(1-F(c))^{pn-k}dc=\int_{c_2}^{c^*_2}F(c)(1-F(c))^{(1-p)n-k}dc$$

[4]Vickrey-Clark-Groves (VCG) auction is a typical sealed-bid auction for multiple items that maximize welfare.

[5]Here we can compute bipartite matching because bidders have unit supply.

But symmetric equilibrium does not work in general because services are heterogeneous and have different feasible request sets (eg. half of the services can only solve the request in A_2). The solved probability is therefore not necessarily "symmetric".

In the above analysis, we are implicitly assuming there is continual arrival of feasible requests so that a unit supply service always chooses one request auction to join without worrying about being idle after solving this request. This suffices if the marketplace is active. But in the case where there might not be a new request for a long period, a service can arrange current requests into compatible bundles and calculate the expected returns from each bundle. By compatible we mean, after a service solving one request, the other request is still feasible for it. Participating in auctions in the optimal bundle would be the ideal procedure to follow.

Same as before, we run a sealed-bid auction with a ceiling price r in the second stage. We have the following theorem. It follows from Theorem 4.1, we provide a proof sketch in Appendix D.

Theorem 4.2. *In l-request auctions with n services as potential bidders where $n \gg l$, $\mathcal{R} = (p, k, \mathbf{x}, \mathbf{p})$ as defined induces optimal entry.*

4.3 Discussion

We discuss here some of the relevant properties and how OpenSquare's mechanism design deals with various scenarios.

Comparison with alternate approach. In Section 1.2, we briefly discussed an alternate approach for outsourcing requests. Translating it to the setting of auctions, we have first the bidders submitting bids and the auctioneer selecting winners. These selected bidders then perform repeated squarings and offer solutions. One adversarial behavior is to bid close to zero to become a winner then crash. In our design, potential bidders compute the solution then submit bids. The malicious parties can still underbid but the client will receive a solution. The negative effect is that honest services might be driven out of the market. Suppose the adversary utilizes k of the services it controls in each attack and has an attacking budget B. If the extra cost of computations while they underbid exceeds B, they stop attacking.

Assume for the worst case, the adversary services maintain the best machine in the system. Consider each service's cost as i.i.d random variables $X_1, \ldots, X_n$ following distribution F. The minimum or the first-order statistics $X_{(1)}$ has PDF $F_{(1)}(x) = 1 - [1 - F(x)]^n$, CDF $f_{(1)}(x) = nf(x)[1-F(x)]^{n-1}$. The expected number of attacks the adversary carries out in the worst case is $\frac{B}{k\int_{\underline{c}}^{\overline{c}} x f_{(1)}(x)dx}$. When the attack is not selective (targeting a specific service), this cost is shared by all other services. Let the adversary control α of the services in the system and let an honest service has an upper bound B' for loss that it can bear with. Then as long as $\frac{B}{(1-\alpha)n} \leq B'$, the attack is not successful. This indicates that if n approaches infinity, B' can be arbitrarily small given that B is finite. But when the attack is selective, then as long as

$$\frac{B\int_{\underline{c}}^{\overline{c}} x f(x)dx}{k\int_{\underline{c}}^{\overline{c}} x f_{(1)}(x)dx} \leq B'$$

the attack is not successful.

To shift or not to shift. In a dynamic setting, new requests arrive. Will a service shift to other auctions in the middle of one auction? This is important because the number of entrants may change and in the worst case, if all services in one auction shift to other auctions, the liveness guarantee may be compromised. This is only meaningful in the multi-request auction setting. In the following scenario, a service S_i considers shifting to other auctions from the current auction A_w: a new request arrives and it enlarges the computations S_i can perform in S_i's current feasibility set by more than the computations it has already spent on A_w. This can happen with a higher probability when S_i just started computations for A_w and the new request fits perfectly into the schedule of other available requests. We argue that not all entrants in the current auction A_w shift due to the winning probability increases in A_w. And since we assumed $n \gg l$, the new request does not induce entrance of all services in a reasonable configuration of p, k.

Buyer incentive to choose higher k while fixing r. In single-request environment, the buyer prefers $k = 1$ since $n \gg 1$. But in a multi-request environment, the request posters are also competing to acquire solutions from services. Given that the user configures (p, k) properly, the probability of not receiving a solution is $[1 - F(c^*)]^{n_0}$, where n_0 is the number of services solving this request and c^* is obtained from the first-order condition (Equation (1)). To obtain a solution with high probability, the key would be to increase n_0 to a certain amount. Intuitively, increasing k directly raises the probability of winning an auction, and increasing budget p enlarges the possible profit from each winning. The configuration of (p, k) and the number of capable servers determine expected profits for bidders. The concrete relationship between them is non-linear but increase k for a fixed r boosts profits, which raises n_0. Overall in a seller's market where demand exceeds supply, the client is incentivized to increase k for a fixed r.

Link to private value. As noted before, in the alternate approach the collateral is directly tied to the client's request value to avoid losses during a DoS attack. In our market equilibrium, demand and supply determine the client's configuration of (p, k). The value of the request indeed affects a client's configuration because it changes c_0 in Equation (1), but more noise is still in play: the request's difficulty, the remaining solving time, and configurations of other requests.

Tractability. The mechanism is tractable in polynomial time because the procurement procedure and determining configurations of p, k take polynomial time.

5 EVALUATION

In this section, we evaluate the OpenSquare protocol by implementing the contract C_{OpSq} and measure the gas costs for the various steps in our protocol presented in Figure 3. While gas costs can in principle be calculated by hand using [46], this is a highly complex task for the case of C_{OpSq} due to its intricate logic. We therefore implement C_{OpSq} in Solidity [3]. As Solidity only supports maps and arrays, in order to maintain the top k asking prices, we use a version of bubble sort, so that we can update our list in $O(n)$ time. This helps save gas costs, and dissuades servers from asking if their asking price is larger than the top k asking prices. Our source code can be found here [4].

5.1 The OpenSquare Smart Contract

In the OpenSquare smart contract C_{OpSq}, we use an RSA modulus of size 2048 for our tests and embed this in our smart contract. To implement the VDF from [44] on Ethereum, and use the code from the public implementations [23].

We present the gas costs for all the functions of C_{OpSq} in Table 1. A limitation of the EVM is that it can only have access to 16 stack variables at any time. This means that we must carefully arrange the variable declarations so that at any point in time, we only access the last 16 local variables. This forces us to split the `NewRequest()` function into two partial functions $\texttt{NewRequest}_1()$ and $\texttt{NewRequest}_2()$. We also implement the logic of issuing complaints, by allowing mini-complaints, where users can complain about specific steps in the verification of a solution. This greatly reduces the gas costs for issuing complaints and in turn reduces the solution collateral V_s and the complaint fee V_c. Wesolowski's VDF [44] makes use of a hash function H_p that hashes to a large prime. The implementation in [23] requires a random nonce value for doing the same, which we assume is available in C_{OpSq}. The contract uses this nonce during a complaint to check the correctness of the hash operation using a single run of the Miller Rabin Primality testing algorithm.

The ask() function is implemented using bubble sort so that all the opened bids are always in increasing order. We estimate the gas cost for the function by considering the worst case for a bubble sort and the plot is shown in Figure 5.

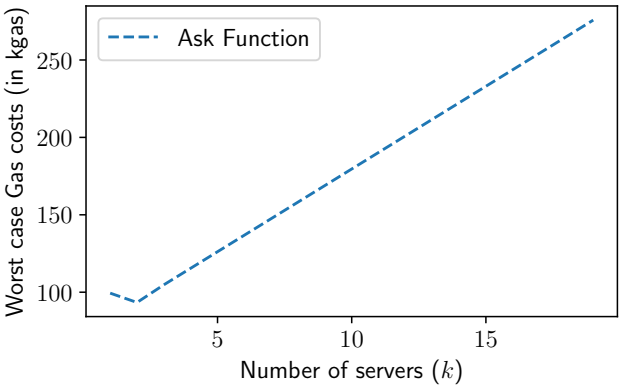

Figure 5: The gas cost of the Ask function, for increasing values of k, which represents the number of servers. These costs are obtained by assuming that the last server to ask is the cheapest (worst case).

Optimisations. In `SubmitSolution()`, we employ an optimization where the server submits x_2, y_1, y_2, and $\pi = (\ell, Q_1, Q_2)$; we compute $h_1 = H(y_1, y_2)$, $h_2 = H(y_1, Q_1)$ and $h_3 = H(y_2, Q_2)$, where H is the Keccak256 hash function; and only store $h = H(h1, h2, h3, pk_S)$. This reduces the storage requirements and also helps in our complaint function implementation.

For implementing `Complaint()`, we split the complaints into independent functions which can trigger one of the following checks, thus minimising computational cost:

(1) Check if ℓ or $\ell + 1$ is computed correctly.

Table 1: Summary of gas costs for the various steps in Figure 3. The numbers have variations up to 1% as Ethereum counts the number of non-zero words and charges accordingly.

Type	Operation	Gas Cost (in gas)
Deployment		4, 996, 164
Optimistic	$\texttt{NewRequest}_1()$	288, 063
	$\texttt{NewRequest}_2()$	275, 463
	`SubmitSolution()`	166, 459
	`Claim()`	85, 470
	`Refund()`	144, 674
Complaints	Invalid prime	102, 521
	Invalid hash to prime	65, 502
	Left/Right check	237, 868

(2) Check if ℓ is a prime.
(3) Check if $Q_1^{\ell} x^r = y_1$ where $r = 2^T \mod \ell$ (Left)
(4) Check if $Q_2^{\ell} x_2^r = y_2$ where $r = 2^T \mod \ell$ (Right)

By splitting the complaint into multiple partial functions, we reduce the complaint fee V_c as the complainant only needs to pay for the computational cost of one of the above checks, and not all.

6 CONCLUSION

In this work we address two main real-world issues with using repeated modular squaring based timed-cryptographic primitives like Time-Lock Puzzles (TLP) and Verifiable Delay Functions (VDF). Specifically, the computational effort in performing the operation and the prediction of the number of squarings to be performed so that security holds in the real world. We address both problems by giving a decentralized repeated modular squaring service protocol OpenSquare. Our game-theoretic analysis shows that our protocol is cost effective and incentive driven. Our protocol can be implemented in practice right away, thus help realize several applications of TLP and VDF. An interesting future direction is to develop sound heuristics for setting the time parameter of these primitives based on the results of running OpenSquare on a system like Ethereum.

ACKNOWLEDGEMENTS

We would like to thank Giulio Malavolta for insightful discussion in the beginning of this work, and Alexandros Psomas for his pointer to unit demand bidders and general comments on formatting a mechanism, and anonymous reviewers for valuable comments.

The work was partially supported by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation) under 442893093, and by the state of Bavaria at the Nuremberg Campus of Technology (NCT). This work has also been partially supported by the Army Research Laboratory (ARL) under grant W911NF-20-2-0026, and the National Science Foundation (NSF) under grant CNS-1846316.

REFERENCES

[1] [n. d.] https://www.vdfalliance.org. ().
[2] [n. d.] https://www.coindesk.com/ethereum-foundation-weighs-15-million-bid-to-build-randomness-tech. ().

[3] [n. d.] (). https://docs.soliditylang.org/en/develop/index.html.

[4] Anonymous. [n. d.] Source code of the contract. https://drive.google.com/file/d/1c4bV_fdxdXRbAAh7HSyr5xPvdiE5cCHs/view?usp=sharing. ().

[5] Iddo Bentov and Ranjit Kumaresan. 2014. How to use bitcoin to design fair protocols. In *CRYPTO 2014, Part II* (LNCS). Juan A. Garay and Rosario Gennaro, editors. Volume 8617. Springer, Heidelberg, (August 2014), 421–439. DOI: 10.1007/978-3-662-44381-1_24.

[6] Vivek Bhattacharya, James W Roberts, and Andrew Sweeting. 2014. Regulating bidder participation in auctions. *The RAND Journal of Economics*, 45, 4, 675–704.

[7] Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. 2016. Time-lock puzzles from randomized encodings. In *ITCS 2016*. Madhu Sudan, editor. ACM, (January 2016), 345–356. DOI: 10.1145/2840728.2840745.

[8] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable delay functions. In *CRYPTO 2018, Part I* (LNCS). Hovav Shacham and Alexandra Boldyreva, editors. Volume 10991. Springer, Heidelberg, (August 2018), 757–788. DOI: 10.1007/978-3-319-96884-1_25.

[9] Dan Boneh and Moni Naor. 2000. Timed commitments. In *CRYPTO 2000* (LNCS). Mihir Bellare, editor. Volume 1880. Springer, Heidelberg, (August 2000), 236–254. DOI: 10.1007/3-540-44598-6_15.

[10] Jeremy Bulow and Paul Klemperer. 1994. Auctions vs. negotiations. Technical report. National Bureau of Economic Research.

[11] Jeremy Bulow and Paul Klemperer. 2009. Why do sellers (usually) prefer auctions? *American Economic Review*, 99, 4, 1544–75.

[12] Benedikt Bünz, Steven Goldfeder, and Joseph Bonneau. 2017. Proofs-of-delay and randomness beacons in ethereum. In.

[13] Ran Canetti. 2001. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 136–145.

[14] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2007. Universally composable security with global setup. In *TCC 2007* (LNCS). Salil P. Vadhan, editor. Volume 4392. Springer, Heidelberg, (February 2007), 61–85. DOI: 10.1007/978-3-540-70936-7_4.

[15] Jing Chen and Silvio Micali. 2017. Algorand. (2017). arXiv: 1607.01341 [cs.CR].

[16] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. 2017. Fairness in an unfair world: fair multiparty computation from public bulletin boards. In *ACM CCS 2017*. Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors. ACM Press, 719–728. DOI: 10.1145/3133956.3134092.

[17] C. Dwork and M. Naor. 2000. Zaps and their applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, 283–293. DOI: 10.1109/SFCS.2000.892117.

[18] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. 2018. FairSwap: how to fairly exchange digital goods. In *ACM CCS 2018*. David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors. ACM Press, (October 2018), 967–984. DOI: 10.1145/3243734.3243857.

[19] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. 2019. Multi-party virtual state channels. In *EUROCRYPT 2019, Part I* (LNCS). Yuval Ishai and Vincent Rijmen, editors. Volume 11476. Springer, Heidelberg, (May 2019), 625–656. DOI: 10.1007/978-3-030-17653-2_21.

[20] Milton Harris and Artur Raviv. 1981. Allocation mechanisms and the design of auctions. *Econometrica: Journal of the Econometric Society*, 1477–1499.

[21] Charles A Holt Jr. 1980. Competitive bidding for contracts under alternative auction procedures. *Journal of political Economy*, 88, 3, 433–445.

[22] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. 2013. Universally composable synchronous computation. In *TCC 2013* (LNCS). Amit Sahai, editor. Volume 7785. Springer, Heidelberg, (March 2013), 477–498. DOI: 10.1007/978-3-642-36594-2_27.

[23] Kilic. [n. d.] Kilic/evmvdf. (). https://github.com/kilic/evmvdf.

[24] Jodie Knapp and Elizabeth A. Quaglia. 2020. Fair and sound secret sharing from homomorphic time-lock puzzles. In *Provable and Practical Security*.

[25] Vijay Krishna. 2009. *Auction theory*. Academic press.

[26] Huijia Lin, Rafael Pass, and Pratik Soni. 2017. Two-round and non-interactive concurrent non-malleable commitments from time-lock puzzles. In *58th FOCS*. Chris Umans, editor. IEEE Computer Society Press, (October 2017), 576–587. DOI: 10.1109/FOCS.2017.59.

[27] Giulio Malavolta and Sri Aravinda Krishnan Thyagarajan. 2019. Homomorphic time-lock puzzles and applications. In *CRYPTO 2019, Part I* (LNCS). Alexandra Boldyreva and Daniele Micciancio, editors. Volume 11692. Springer, Heidelberg, (August 2019), 620–649. DOI: 10.1007/978-3-030-26948-7_22.

[28] Eric Maskin and John Riley. 2000. Optimal multi-unit auctions'. *International library of critical writings in economics*, 113, 5–29.

[29] R Preston McAfee and John McMillan. 1987. Auctions with entry. *Economics Letters*, 23, 4, 343–347.

[30] Ahmet Can Mert, Erdinc Ozturk, and Erkay Savas. 2020. Low-latency asic algorithms of modular squaring of large integers for vdf evaluation. *IEEE Transactions on Computers*, 1–1. DOI: 10.1109/TC.2020.3043400.

[31] [n. d.] Minimal vdf randomness beacon. https://ethresear.ch/t/minimal-vdf-randomness-beacon/3566. ().

[32] Diego Moreno and John Wooders. 2011. Auctions with heterogeneous entry costs. *The RAND Journal of Economics*, 42, 2, 313–336.

[33] Roger B Myerson. 1989. Mechanism design. In *Allocation, Information and Markets*. Springer, 191–206.

[34] Roger B Myerson. 1981. Optimal auction design. *Mathematics of operations research*, 6, 1, 58–73.

[35] Krzysztof Pietrzak. 2019. Simple verifiable delay functions. In *ITCS 2019*. Avrim Blum, editor. Volume 124. LIPIcs, (January 2019), 60:1–60:15. DOI: 10.4230/LIPIcs.ITCS.2019.60.

[36] R. L. Rivest, A. Shamir, and D. A. Wagner. 1996. Time-lock Puzzles and Timed-release Crypto. Technical report. Cambridge, MA, USA.
[37] James W Roberts and Andrew Sweeting. 2013. When should sellers use auctions? *American Economic Review*, 103, 5, 1830–61.
[38] William F Samuelson. 1985. Competitive bidding with entry costs. *Economics letters*, 17, 1-2, 53–57.
[39] Philipp Schindler, Aljosha Judmayer, Markus Hittmeir, Nicholas Stifter, and E. Weippl. 2020. Randrunner: distributed randomness from trapdoor vdfs with strong uniqueness. *IACR Cryptol. ePrint Arch.*, 2020, 942.
[40] Sri Aravinda Krishnan Thyagarajan, Adithya Bhat, Giulio Malavolta, Nico Döttling, Aniket Kate, and Dominique Schröder. 2020. Verifiable timed signatures made practical. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (CCS '20). Association for Computing Machinery, Virtual Event, USA, 1733–1750. ISBN: 9781450370899. DOI: 10.1145/3372297.3417263. https://doi.org/10.1145/3372297.3417263.
[41] Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, Fritz Schmidt, and Dominique Schröder. 2020. Paymo: payment channels for monero. Cryptology ePrint Archive, Report 2020/1441. https://eprint.iacr.org/2020/1441. (2020).
[42] [n. d.] Vdf fpga competition round 2 results. https://github.com/supranational/vdf-fpga-round2-results. ().
[43] [n. d.] Verifiable delay functions and attacks. https://ethresear.ch/t/verifiable-delay-functions-and-attacks/2365. ().
[44] Benjamin Wesolowski. 2019. Efficient verifiable delay functions. In *EUROCRYPT 2019, Part III* (LNCS). Yuval Ishai and Vincent Rijmen, editors. Volume 11478. Springer, Heidelberg, (May 2019), 379–407. DOI: 10.1007/978-3-030-17659-4_13.
[45] Benjamin Wesolowski and Ryan Williams. 2020. Lower bounds for the depth of modular squaring. Cryptology ePrint Archive, Report 2020/1461. https://eprint.iacr.org/2020/1461. (2020).
[46] Gavin Wood et al. 2014. Ethereum: a secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151, 1–32.
[47] Lixin Ye. 2007. Indicative bidding and a theory of two-stage auctions. *Games and Economic Behavior*, 58, 1, 181–207.

A MORE PRELIMINARIES

Time-Lock Puzzles. We recall the definition of standard time-lock puzzles [7]. For conceptual simplicity we consider only schemes with binary solutions.

Definition A.1 (Time-Lock Puzzles). A time-lock puzzle is a tuple of two algorithms $(\mathsf{PGen}, \mathsf{PSolve})$ defined as follows.

- $Z \leftarrow \mathsf{PGen}(\mathbf{T}, m)$: the puzzle generation algorithm takes as input a hardness-parameter $\mathbf{T}$ and a solution $m \in \{0,1\}$, and outputs a puzzle Z.
- $m \leftarrow \mathsf{PSolve}(Z)$: the puzzle solving algorithm is a deterministic algorithm that takes as input a puzzle Z and outputs a solution m.

Definition A.2 (Correctness). For all $\lambda \in \mathbb{N}$, for all polynomials $\mathbf{T}$ in λ, and for all $m \in \{0,1\}$, it holds that $m = \mathsf{PSolve}(\mathsf{PGen}(\mathbf{T}, m))$.

Definition A.3 (Security). A scheme $(\mathsf{PGen}, \mathsf{PSolve})$ is secure with gap $\varepsilon < 1$ if there exists a polynomial $\tilde{\mathbf{T}}(\cdot)$ such that for all polynomials $\mathbf{T}(\cdot) \geq \tilde{\mathbf{T}}(\cdot)$ and every polynomial-size adversary $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda\in\mathbb{N}}$ of depth $\leq \mathbf{T}^\varepsilon(\lambda)$, there exists a negligible function $\mathsf{negl}(\lambda)$, such that for all $\lambda \in \mathbb{N}$ it holds that

$$\Pr\left[b \leftarrow \mathcal{A}(Z) : \quad Z \leftarrow \mathsf{PGen}(\mathbf{T}(\lambda), b) \quad \right] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

Verifiable Delay Functions. We recall the formal definition of the verifiable delay functions from [8].

Definition A.4. A *verifiable delay function* (VDF) scheme, consists of a tuple of 4 PPT algorithms $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ that are defined below:
$pp \leftarrow \mathsf{Setup}(1^\lambda, \mathbf{T})$: the setup algorithm takes as input a security parameter 1^λ and a time parameter $\mathbf{T}$, and outputs the public parameters pp. The public parameters encode an input domain $\mathcal{X}$ and output domain $\mathcal{Y}$.
$x \leftarrow \mathsf{Gen}(pp)$: the instance generation algorithm takes as input the public parameters, and internally samples $x \leftarrow \mathcal{X}$ to output x.
$(y, \pi) \leftarrow \mathsf{Eval}(pp, x)$: the evaluation algorithm takes as input the public parameters pp, an instance x, and outputs a result $y \in \mathcal{Y}$, and a proof π.
$0/1 \leftarrow \mathsf{Verify}(pp, x, y, \pi)$: the verification algorithm takes as input the public parameters pp, an instance x, a result y, and a proof π, and outputs 1 if the result y is valid with respect to x and outputs 0 otherwise.

Definition A.5 (Completeness). A verifiable delay function scheme Π_{VDF} $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ is said to be complete if for all $\lambda, \mathbf{T} \in \mathbb{N}$, the following holds:

$$\Pr\left[\mathsf{Verify}(pp, x, y, \pi) = 1 \middle| \begin{array}{c} pp \leftarrow \mathsf{Setup}(1^\lambda, \mathbf{T}) \\ x \leftarrow \mathsf{Gen}(pp) \\ (y, \pi) \leftarrow \mathsf{Eval}(pp, x) \end{array}\right] = 1.$$

Definition A.6 (Sequentiality). A verifiable delay function scheme $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ is said to be sequential if for all $\lambda, \mathbf{T} \in \mathbb{N}$, all pairs of PPT adversaries $(\mathcal{A}_1, \mathcal{A}_2)$ such that the parallel running time of $\mathcal{A}_2$ is is bounded by $\mathbf{T} \in \mathbb{N}$, there exists a negligible function negl such that

$$\Pr\left[(y, \cdot) \leftarrow \mathsf{Eval}(pp, x) \middle| \begin{array}{c} pp \leftarrow \mathsf{Setup}(1^\lambda, \mathbf{T}) \\ (\mathsf{st}) \leftarrow \mathcal{A}_1(pp) \\ x \leftarrow \mathsf{Gen}(pp) \\ (y, \pi) \leftarrow \mathcal{A}_2(\mathsf{st}, x) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Definition A.7 (Soundness). A verifiable delay function scheme $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ is said to be sound, if for all $\lambda, \mathbf{T} \in \mathbb{N}$, all PPT adversaries $\mathcal{A}$, there exists a negligible function negl, such that

$$\Pr\left[\begin{array}{l} (y, \cdot) \neq \mathsf{Eval}(pp, x) \wedge \\ \mathsf{Verify}(pp, x, y, \pi) = 1 \end{array} \middle| \begin{array}{c} pp \leftarrow \mathsf{Setup}(1^\lambda, \mathbf{T}) \\ (x, y, \pi) \leftarrow \mathcal{A}(pp) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Functionality $\mathcal{L}$, running with a set of parties $P_1, \ldots, P_n$ stores the balance $p_i \in \mathbb{N}$ for every party P_i , where $i \in [n]$ and a partial function K for frozen cash. It accepts queries of the following types:

Update funds: Upon receiving message (update, P_i, p) with $p \geq 0$ from $\mathcal{E}$, set $p_i := p$ and send $(\text{updated}, P_i, p)$ to every entity.

Freeze funds: Upon receiving message $(\text{freeze}, \text{id}, P_i, p)$ from an ideal functionality of session id check if $p_i > p$. If this is not the case, reply with $(\text{nofunds}, P_i, p)$. Otherwise set $p_i := p_i - p$, store (id, p) in K and send $(\text{frozen}, \text{id}, P_i, p)$ to every entity.

Unfreeze funds: Upon receiving message $(\text{unfreeze}, \text{id}, P_j, p)$ from an ideal functionality of session id, check if $(\text{id}, p') \in K$ with $p' \geq p$. If this check holds update (id, p') to $(\text{id}, p' - p)$, set $p_j := p_j + p$ and send $(\text{unfrozen}, \text{id}, P_j, p)$ to every entity

Figure 6: Global ledger functionality $\mathcal{L}$ [18].

Universal Composability. Our security model for OpenSquare is in the *universal composability* framework from Canetti [13]. We also consider the extension to a global setup [14], that helps model concurrent executions. We consider *static* corruptions, where the adversary announces at the beginning which parties he corrupts. We denote the environment by $\mathcal{E}$. For a real protocol Π and an adversary $\mathcal{A}$ we write $EXEC_{\tau,\mathcal{A},\mathcal{E}}$ to denote the ensemble corresponding to the protocol execution. For an ideal functionality $\mathcal{F}$ and an adversary $\mathcal{S}$ we write $EXEC_{\mathcal{F},\mathcal{S},\mathcal{E}}$ to denote the distribution ensemble of the ideal world execution.

Definition A.8 (Universal Composability). A protocol τ UC-realizes an ideal functionality $\mathcal{F}$ if for any PPT adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for any environment $\mathcal{E}$ the ensembles $EXEC_{\tau,\mathcal{A},\mathcal{E}}$ and $EXEC_{\mathcal{F},\mathcal{S},\mathcal{E}}$ are computationally indistinguishable.

We make use of a ledger functionality $\mathcal{L}$ described in Figure 6 that is borrowed from [18]. The functionality lets parties transfer coins between them while allowing coins to be locked and unlocked at contracts. The functionality has a partial function $K : \{0,1\}^* \rightarrow \mathbb{N}$ that maps a contract identifier id to an amount of coins that is locked for the execution of contract id. The parties cannot directly interact with the functionality, but other ideal functionalities can adjust balances of parties based on freeze and unfreeze messages. Freeze messages, transfer coins from the balance of a party to a contract, while unfreeze message, transfers the frozen coins back to the user's account.

We assume synchronous communication between parties, where the execution of the protocol happens in rounds. We model this via an ideal functionality called the clock functionality denoted by $\mathcal{F}_{\text{clock}}$ from [22, 19], where all honest parties are required to indicate that are ready to proceed to the next round before the clock proceeds. The exact clock functionality that we consider is fully described in [14]. In the functionality all entities are always aware of the given round and users can abort a session at any given round by sending a an abort message.

B UC FORMULATION OF A DECENTRALIZED SOLUTION SERVICE

We present here the security formulation of a decentralized solution service in the form of an ideal functionality. The goal is to model our OpenSquare protocol but abstracting away the specific computational details.

The main property that a solution service must guarantee is a fair exchange of digital goods: a solution to a problem ϕ and p coins as reward. More precisely, there are services $S_1, \ldots, S_n$ and a client C. The client has a problem ϕ he wishes to get solved and is willing to offer certain price of up to p coins according to a certain distribution $\mathcal{R}$. The services may propose solutions to the problem and ask for a certain price for their service. Finally the services are rewarded according to their asking price and the reward distribution $\mathcal{R}$.

Dziembowski, Eckert and Faust [18] gave a UC formulation of a similar exchange where there is a single seller willing to sell a witness w for the circuit ϕ, in exchange for a price p from the buyer. However, their formulation falls short in certain crucial aspects for the decentralized solution notion we want. It is immediate to see that in our formulation we have multiple services (or sellers) whereas their setting is tailored for a single seller. Secondly, the price p for the exchange is fixed ahead and known to both the seller and the client in their formulation. Whereas, in the case of decentralized solution service, the services do not yet know the solution to the problem, and therefore have to perform some computation to generate the solution. Only after the computation can the services determine their price and they quote the asking price. Thirdly, in our setting, depending on the problem ϕ and the reward distribution $\mathcal{R}$, services can decide to continue or abort in attempting to solve the problem. This is in contrast to their setting where both the seller and the client interact with the functionality having agreed to the price p ahead of time. Similar issues persist with other formulations of blockchain based cryptographic fairness [16, 5].

B.1 Ideal Functionality

The formal description of our functionality $\mathcal{F}_{\text{SolS}}$ is given in Figure 7. The functionality captures decentralized solving service between a client C and n services $S_1, \ldots, S_n$. On a high level, a client sends a request to the functionality. The request specifies the problem, and the reward distribution. The client is also required to lock a reward amount to the ideal functionality. The services can send solutions to the functionality along with their asking price. Finally, valid solutions are rewarded based on the reward distribution chosen by the client and the asking price quoted by the service for the solution. Remaining coins are refunded back to the client.

In more detail, during the request phase, the client sends a request $(\text{req}, \text{id}, \phi, R)$ where $R = (p, \text{addr}_C, \mathcal{R})$. Here ϕ is the computational problem modelled here as a circuit $\phi : \{0,1\}^* \rightarrow \{0,1\}$, the reward amount is p, the address of the client is addr_C and the reward distribution function $\mathcal{R} : \{0,1\}^* \rightarrow \{0,1\}^*$. The function $\mathcal{R}$ is publicly verifiable, and we elaborate on this later. The functionality freezes the p coins at an address of the functionality specific to the session. The functionality also sends the request and the reward distribution function to all the services.

During the service phase, services can either abort or send a solution $(\text{Sol}, \text{id}, s, a, \text{addr})$ where s is the solution, a is the asking price

and addr is the address of the service. The functionality records the solution in a list L_{id} specific to the session with identifier id. After receiving messages from every service, the functionality returns the asking prices of all the services to the simulator.

During the payout phase, the functionality eliminates all the invalid solutions by checking if $\phi(s, \mathsf{addr}) = 0$, where s is the solution and addr is the address of the service that registered the solution. The functionality then uses the reward distribution function $\mathcal{R}$ with inputs all the valid solution and the reward price p. The function returns a set $\{(i, p_i, \mathsf{addr}_i)\}_{i\in[|L_{\mathsf{id}}|]} \leftarrow \mathcal{R}(L_{\mathsf{id}}, p)$, containing the index of the service, the price to be paid p_i and the address of the service addr_i to which the price would be paid. The functionality unfreezes the coins from the p coins that was frozen, and transfers these coins to the services, that is, p_i coins are transferred to addr_i of service S_i. The remaining coins after transferring the rewards to services, are transferred to the client. If there was no valid solution, all the p coins are refunded to the client.

Fairness and Public Verifiablity of $\mathcal{R}$. The functionality ensures fairness in that the services receive rewards according to the reward distribution $\mathcal{R}$ as specified by the client and known to the services, if and only if the service recorded a valid solution to the problem ϕ. Moreover, provided the reward distribution function $\mathcal{R}$ returns reward prices such that $\sum_{i\in[|L_{\mathsf{id}}|]} p_i \leq p$ and this is verifiable by the services during the request phase, we are guaranteed that the reward amount p is enough to reward all the solutions. If this property of $\mathcal{R}$ is not publicly verifiable, services can simply abort during the service phase. This ensures that the services can be rewarded correctly according to $\mathcal{R}$ from the reward p if they record a valid solution.

The ideal functionality $\mathcal{F}_{\mathsf{SolS}}$ (in session id) interacts with a client C, services $S_1, \ldots, S_n$, the ideal adversary $\mathcal{S}$ and the global ledger $\mathcal{L}$.

Request Phase

- Upon receiving $(\mathsf{Req}, \mathsf{id}, \phi, R)$ with $R := (p, \mathsf{addr}_C, \mathcal{R})$ and $p \in \mathbb{N}$ from C, leak $(\mathsf{Req}, \mathsf{id}, \phi, R)$ to $\mathcal{S}$, store the circuit ϕ and the reward R.
- Initialize a list $L_{\mathsf{id}} := \emptyset$
- Send $(\mathsf{freeze}, \mathsf{id}, C, p)$ to $\mathcal{L}$. If the response is $(\mathsf{frozen}, \mathsf{id}, C, p)$, send $(\mathsf{Req}, \mathsf{id}, \phi, R)$ to all services $(S_1, \ldots, S_n)$, and then go to service phase.

Service Phase

- Upon receiving $(\mathtt{abort}, \mathsf{id})$ from all of the services, then send $(\mathsf{unfreeze}, \mathsf{id}, C, p)$ to $\mathcal{L}$ and terminate.
- Otherwise, upon receiving $(\mathsf{Sol}, \mathsf{id}, s_i, a_i, \mathsf{addr}_i)$ from service S_i, register $L_{\mathsf{id}} := L_{\mathsf{id}}||(i, s_i, a_i, \mathsf{addr}_i)$, and leak (i, s_i) to $\mathcal{S}$.
- At the end of the phase, leak all a_i values obtained in this phase to $\mathcal{S}$, and accept no more messages with Sol.

Payout Phase

- For $k \in [|L_{\mathsf{id}}|]$, parse $L_{\mathsf{id}}[k] := (i, s, a, \mathsf{addr})$ check if $\phi(s, \mathsf{addr}) = 1$, if so do nothing. Otherwise, remove (i, s, a, addr) from the list L_{id}.
- If $L_{\mathsf{id}} \neq \emptyset$, compute $\{(i, p_i, \mathsf{addr}_i)\}_{i\in[|L_{\mathsf{id}}|]} \leftarrow \mathcal{R}(L_{\mathsf{id}}, p)$. Send $(\mathsf{unfreeze}, \mathsf{id}, S_i, p_i)$ for all $i \in [|L_{\mathsf{id}}|]$ and $(\mathsf{unfreeze}, \mathsf{id}, C, p - \sum_{i\in[|L_{\mathsf{id}}|]} p_i)$ to $\mathcal{L}$ and send $(\mathsf{sold}, \mathsf{id}, \{s_i\}_{i\in[|L_{\mathsf{id}}|]})$ to the client C.
- If $L_{\mathsf{id}} = \emptyset$, then send $(\mathsf{unfreeze}, \mathsf{id}, C, p)$ to $\mathcal{L}$ and send $(\mathsf{unsold}, \mathsf{id})$ to the client C.

Figure 7: Ideal functionality $\mathcal{F}_{\mathsf{SolS}}$ for solution services.

C SECURITY ANALYSIS

We prove the following theorem that formally states the security of our OpenSquare protocol.

Theorem C.1. *Let* $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ *be a watermarked VDF that is sequential and sound. Then our OpenSquare protocol run between a client, and n servers with access to the smart contract* C_{OpSq}*, a global ledger* $\mathcal{L}$ *(Figure 6), and a clock functionality* $\mathcal{F}_{\mathsf{clock}}$*, UC-realizes the functionality* $\mathcal{F}_{\mathsf{SolS}}$*.*

Proof sketch of Theorem C.1. Since there is no secrecy requirement, our simulation strategy is fairly simple. The simulator is the ideal world adversary that interacts with the functionality $\mathcal{F}_{\mathsf{SolS}}$ and simulates the view of the adversary $\mathcal{A}$. We have two cases, where either the client is honest or a server is honest.

In the later case, assume that all entities except server S_i are corrupt. In this case the simulator simply relays the messages between the adversary and the ideal functionality.

In the former case, we describe the simulator by making use of two hybrid executions. The first hybrid execution is that of the real world protocol.

The second hybrid, is the same as the first hybrid except that the simulation aborts if the adversary posts a solution $(\mathtt{RspID}, y, \pi, h)$ for a request $\mathsf{req} := (g, \mathbf{T}, N)$ from a public key pk such that $\mathsf{Verify}((\mathbf{T}, N), y, \pi, pk) = 1$, and $\mathsf{Eval}((\mathbf{T}, N), g, pk) \neq y$.

The execution in the first and the second hybrid are indistinguishable. Notice that the only difference between the hybrids is the event that an abort happens in the second hybrid. But the probability with which the abort event happens in the second hybrid can be bound by a negligible function following from the soundness of the watermarked VDF.

□

D AUCTION

D.1 Single request auction

Theorem 4.1. *In single-request auction with n services as potential bidders where* $n \gg 1$*,* $\mathcal{R} = (p, k, \mathbf{x}, \mathbf{p})$ *as defined induces optimal entry.*

Remark 1. *Here by "optimal", we mean the auction achieves minimized costs. The entry induced by the configuration of* (p, r) *is as desired by the client.*

Proof. We set unit ceiling price r in such a way that in equilibrium, the expected buyer cost for k solutions is minimized. Note that we do not vary n, the number of potential bidders, after the derivation. So we do not discuss the potential effects brought by a greater or smaller n. Since services have unit demand and cost distribution F is regular, the expected buyer cost is minimized by procuring up to k solutions from up to k services with the lowest bids below the unit ceiling price.

Additionally, we state the revenue equivalence theorem in the standard auction context. Recall that reverse auction with ceiling price has mathematically equivalent forward auction form.

Theorem D.1 (Revenue Equivalence Theorem). *In the IPV setting, any two auctions in which the following hold in equilibrium: (1) the bidder with the highest valuation wins the auction; (2) any bidder with the lowest possible valuation pays 0 in expectation. Then the expected payoffs to each type of each bidder, and the seller's expected revenue are the same in both auctions.*

One example is a Bayesian Nash Incentive Compatible variation of first price auction and second price auction. Harris and Raviv [20] show that Revenue Equivalence Theorem continues to hold for unit demand bidders in multi-unit auctions for uniform distribution of bidders' valuations. Maskin [28] shows it for general distributions of valuations as long as regularity assumption holds.

Theorem D.2 (Revenue Equivalence Theorem, multi-unit). *Assume that each of n risk-neutral agents has an independent private valuation for a single unit of k identical goods at auction, drawn from a common cumulative distribution $F(v)$ that is strictly increasing and atomless on $[\underline{v}, \overline{v}]$. Then any efficient auction mechanism in which any agent with valuation $\underline{v}$ has an expected utility of zero yields the same expected revenue, and hence results in any bidder with valuation v_i making the same expected payment.*

In our second-stage auction, the bidders with the lowest costs win the auction since bidding function $b(c)$ is monotone non-decreasing in c and we pick the lowest k bidders below ceiling price as winners. A bidder with costs higher than the ceiling price or higher than the winners has utility 0. Revenue equivalence theorem applies. Maskin also provides detailed proof for the optimality of a selling procedure, selling up to k units to buyers with the highest bidding price above a reserved price, for multi-unit auction in [28, Proposition 4].

□

D.2 Multi-request Auction

Theorem 4.2. *In l-request auctions with n services as potential bidders where $n \gg l$, $\mathcal{R} = (p, k, \mathbf{x}, \mathbf{p})$ as defined induces optimal entry.*

proof sketch. Potential bidders have unit demand, so we only need to show that there is a sufficient number of bidders for an auction, and then we can consider a single-request auction environment. If a request is not feasible for the system, we do not guarantee a solution. If a request is feasible, then by considering the number of capable services and solving for the ceiling price, the user can attract the desired amount of potential bidders to the auction.

Let $A_1, \ldots, A_l$ be the l auctions with ascending break-even prices. The number of services with costs lower than each break-even price (interested in entry) is also ascending. In the procedure of configuring an auction A_w, the client takes into account the opportunity costs of not attending the best auction A_{best} other than A_w and covers the expected returns in A_{best}. This means that this auction is better than A_{best} in its full load (all capable services participate in it). We show that with high probability, at least one capable service bid in A_w.

A service S_i with cost c_{iw} for auction A_w calculates the profits from participating in auction A_w as $\pi_w(c_{iw}) = (r_w - c_{iw})[1 - F(c_{iw})]^{n_w - k_w}$, where n_w increases with w. For auction A_w to have no bidders, we need $\forall j \neq w, \pi_j(c_{ij}) = (r_j - c_{ij})[1 - F(c_{ij})]^{n_j - k_j} > (r_w - c_{iw})$. We know from the approach the client uses to determine the auction that $\pi_j(c_{ij}) < \pi_w(c_{iw})$ at their full load. This means that $\pi_j(c_{ij}) < (r_w - c_{iw})$. Since we assume $n \gg l$, by law of large numbers, we know the costs drawn by the n services are close to the mean value with high probability and $n_w, n_j \gg 1$ with high probability (the probability of $n_w = \mathbf{s}(w)F(c^*_{iw}) \gg 1$, $n_j = \mathbf{s}(j)F(c^*_{ij}) \gg 1$). Therefore, this reasoning can apply to all such auction A_j including the A_{best} and still have at least one bidder in this auction A_w with high probability.

Note that if there's no existing auction to compare to when a client starts the request, we go back to the single-request auction environment. □