Collaborative Verifiable Delay Functions

No Author Given

No Institute Given

Abstract. We propose and define a new primitive, collaborative verifiable delay functions (coVDFs), an extension to VDFs allowing multiple parties to jointly compute a publicly verifiable delay, whilst encapsulating a personal input from each party. These personal inputs can contain information such as a hash of a bid in an auction, or some public identifier of the solving party. We highlight the differences between the single-party and the multi-party settings, and discuss some applications facilitated by the additional properties offered by coVDFs.

We formalise this new primitive, as well as the relevant security properties, and we introduce the notions of *robustness* and *traceability*, which mitigate adversarial behaviour arising from the introduction of multiple parties. We propose two candidate constructions: the first is an extension of Wesolowski's VDF construction from EUROCRYPT 2019, relying on repeated squaring in a finite abelian group. We prove that this extension satisfies the traceability property, however it is not robust. Our second construction is based on the hashgraph protocol proposed by Baird in 2016, and involves each of the n parties repeatedly implementing a gossip protocol to one another and computing a hash each time they do. The construction results in every party producing a copy of the same graph, and is robust, meaning it runs correctly, in the presence of up to $n/3$ malicious parties.

Keywords: Verifiable Delay Functions, Repeated Squaring, Hashgraph

1 Introduction

In recent years, the rapid growth of distributed ledger technology has fuelled research into proving effort has been expended. The most well-known case is the proof of work system used by Bitcoin [31], among other blockchains. However, this resource-intensive design has received widespread criticism due to its vast energy consumption and the associated impact on climate change [37]. Alternative, more energy efficient methods of showing that computational effort has been expended have been proposed [29, 14, 22, 13], among them verifiable delay functions (VDFs) [11, 16, 17, 12].

VDFs, first formalised in [11] in 2018, have quickly become a very active research area [38, 32, 17, 24]. They are used to show that some amount of clock time has elapsed, by running a sequential algorithm resistant to massive parallelisation: regardless of how many cores each participant has access to, they must be able to reach the same solution at approximately the same time. VDFs enable a variety of applications including building resource efficient blockchains, public randomness beacons and timestamping mechanisms [14, 24, 11].

More formally, a VDF consists of a triple of algorithms, Setup, Eval and Verify. Setup takes security parameter λ and delay parameter t, and outputs public parameters which specify how to compute and verify the VDF. Eval takes an input and the public parameters, and provides an output y and a proof π. Verify uses π to efficiently verify that y is the correct output for x, implying that t steps have been calculated. A VDF must be sequential, efficiently verifiable, and have a unique output for any input. The sequentiality property links this computation to wall-clock time, by requiring that each step of the computation relies on the output of the previous step. This prevents the entire computation being parallelised, and is how a delay is achieved. Uniqueness

ensures that work from one VDF instance cannot be re-used in another computation to circumvent the delay.

In this work, we extend VDFs to a group setting and define a collaborative verifiable delay function (coVDF), where n parties jointly compute the delay function. We additionally allow each party to embed a *personal input* into the computation. This gives the primitive additional functionality, allowing the delay to be utilised in a wider array of settings. A personal input can be a commitment to a vote or a bid, or it could be some public information showing the age of a document, or an indicator of how much work has been contributed by a single party.

Furthermore, our new primitive allows us to take a VDF application such as timestamping and 'batch' it, allowing multiple parties to timestamp documents together in a way that is either synchronised or sequential. That is, we can either show multiple documents to all be the same age, or obtain an ordering of such documents, where documents are added incrementally. We discuss applications in greater detail in Section 1.2.

1.1 Related Work

VDFs were first introduced by Boneh et al. in 2018 [11], motivated by applications in decentralized systems, such as public randomness beacons, leadership elections in consensus protocols, and timestamping.

Three candidate VDF schemes were proposed soon after [38, 32, 17], based on repeated squaring in a group of unknown order, and on isogenies over elliptic curves. An intuitive approach to building a coVDF could be to use a single-party VDF and *split* the work between each of the n solvers. In practice, this is not necessarily easy as output proofs also need to be split, and some method is required to add personal inputs into the protocol. In Section 3, we show that this intuition does work in specific cases, and, in particular, we extend Wesolowski's VDF [38] to a coVDF, giving a concrete instantiation from repeated squaring.

In EUROCRYPT 2020, Ephraim et al. introduced the notion of a continuous VDF (cVDF) [16], defined as a VDF which can be verified at regular intervals rather than at the end of the computation. This allows standard VDF applications such as timestamping or providing a randomness beacon to be outsourced, and also allows for another party to take over the computation at any verification point. Whilst cVDFs are similar to (a specific class of) coVDFs, the key difference is that coVDFs allow users to include a personal input, enabling more functionalities.

Extending a VDF to a multi-user setting can be seen as closely related to Multi-Party Computation (MPC), which allows evaluation of an arbitrary function on private inputs from multiple parties without revealing the inputs [33]. In a coVDF, a group of parties jointly provide a set of private inputs, with a computational timestamp and proof of effort associated with the primitive's computation. One typical application of secure MPC is blind auctions, which we discuss in the next section as a potential application of coVDFs.

1.2 Applications of a coVDF

We next explore some use cases of coVDFs, distinguishing between two distinct classes of coVDFs, namely *sequential* and *parallel*, for which we provide a formal definition (Definition 2) in Section 2. Intuitively, in a sequential coVDF parties provide their personal input at various stages in the computation, while parallel coVDFs require parties to provide their personal input at the same time, i.e., at the beginning of the computation. This leads to separate applications arising in each

setting, which we expand upon next.

Applications of sequential coVDFs. In a sequential coVDF, parties take turns to calculate their part of the computation, before passing it on to another party to continue.

Sequential coVDFs can find a natural application in the setting of collaborative work. Consider an entity such as an employer or teacher, who wants a group (such as employees or students) to complete some collaborative activity, which requires effort from all parties, but each part of this activity relies on the previous one being completed. In such a scenario, the use of a sequential coVDF can be used to provide evidence that the work has been done by each party, as well as providing the next party the required information to start their task. Alternatively, a coVDF can be used by a group of parties as a proof of expended effort, in return for access to some resource owned by a company. In particular, this can be done by parties who can only expend effort at certain times, for example at night when electricity is cheaper [5, 21]. Upon verification, the company can then use this computation to implement a randomness beacon. In Appendix A, we outline how to achieve this using our sequential coVDF construction from Section 3.

Applications of parallel coVDFs. Using a parallel coVDF, n parties each submit a personal input, and together calculate a delay on this set of inputs. The knowledge that all parties submitted these inputs together, and before some particular event, can be useful in various settings. The example we focus on is decentralised blind auctions.

In a blind auction, each party submits a sealed bid, such that no bidder knows the bid of any other participant. The party with the highest bid wins the auction, and pays the price they submitted for the goods [15]. Blind auctions have been subject to recent study in the decentralised setting, for example [19] and [2] propose auctioneer-free sealed-bid protocols, using smart contracts on Ethereum and multi-party computation, respectively. However, in schemes such as those listed above, parties post their bids on a public bulletin board. The drawback of this is that those who bid later have knowledge of how many bids have been submitted, which can be construed as an advantage over those who bid first. This can also mean that bidders bid higher: when bidders have constant absolute risk aversion, the expected selling price in such auctions is higher when the bidders do not know how many other bidders there are, compared with when they do know this [30]. These two points provide an argument that an auction where all parties know n is fairer for the bidders.

coVDFs can be used to instantiate fair, blind auctions in a decentralised setting, in such a way that no private communication channels are required and all results are publicly verifiable. Running a coVDF will give each party an opportunity to bid simultaneously, whilst knowing n, and hence providing a fairer system. In Section 4, we provide a construction for a parallel coVDF, and in Appendix A, we show explicitly how our construction can be used in the context of blind auctions.

1.3 Our contributions

In this paper we propose and formally define a collaborative VDF, a primitive in which a fixed number of parties n take an external input and a personal input and spend a certain amount of time t computing an output, using sequential calculations. Once this computation has been completed, it can be verified as correct by anybody in time $O(\text{polylog}(t))$. We define the security notions of correctness, soundness and sequentiality for coVDFs.

When moving to a trustless multi-party setting, one must consider malicious parties who wish to abort the protocol in order to learn some information [1, 27]. To mitigate this problem, we introduce two properties: *traceability* - which allows malicious parties to be identified and removed from the protocol, allowing a successful rerun; and *robustness* - capturing the idea that the protocol still runs correctly even when some fraction of n is malicious. We model these properties formally in the context of our new primitive.

We propose two constructions: for our first construction, we show how to adapt an existing single-party VDF scheme [38], which is based on repeated squaring in a group of unknown order, to the multi-party setting. We describe a method for obtaining traceability, and prove its security.

Our second construction is based on gossip and sequential hashing, and satisfies robustness. For this construction we borrow techniques from the hashgraph consensus protocol proposed in [3] [1]. Each hash, known as an event, is recorded in a graph. The n solvers repeatedly implement a gossip protocol [9], in which they "gossip about gossip": each time a party gossips to another, they tell the receiving party all the new events that they became aware of since the last sync between the two parties. The receiving party creates an event to represent this new sync, by hashing the most recent event from both parties. In this process, information spreads exponentially quickly through the n solvers, and each party ends up with a graph that is *consistent* with all other parties. We use the techniques of [3] to ensure that all parties obtain a copy of the same graph, and to show that our construction is robust, providing we have a 2/3 honest majority.

2 coVDFs: Definitions and Security Properties

A collaborative verifiable delay function (coVDF) allows users to jointly compute a delay, with the option of embedding a personal input. We begin by providing a formal definition for coVDFs, before categorising constructions as sequential or parallel. We define a coVDF, as well as the properties of *correctness*, *soundness* and *sequentiality* using syntax which covers both of these cases.

Notation In what follows, we refer to two different types of input for each party: We denote the external input of party i by c_i, which is the standard input used to run the computation. This may come from another solving party, or from the third party who generates the instance. In particular, c_0 is the seed used at the start of each instance, and this should be generated independently of all solving parties. We denote by x_i the optional personal input as described in Section 1.2. We use N to refer to the set of all n solving parties. Additionally, we use H to denote a collision resistant hash function.

Definition 1 (coVDF). *A* collaborative verifiable delay function V = *(Setup, Eval, Verify) consists of the following triple of algorithms, implemented by an initiator, n solvers, and a verifier:*

$\mathsf{Setup}(\lambda, t, n) \to \mathbf{pp} = (\mathsf{ek}, \mathsf{vk})$ is ran by the initiator, taking a security parameter λ, delay parameter t, and the number of solvers n as inputs. Setup returns the following public parameters: an evaluation key ek and the corresponding verification key vk. Where appropriate, the public parameters also specify the input space $\mathcal{X}$ and the output space $\mathcal{Y}$.

[1] Baird introduced the Hashgraph in 2016 as a fair, fast consensus protocol. It has since been used as the basis for a cryptocurrency, Hedera Hashgraph [4].

$\mathsf{Eval}(\mathsf{ek}, c_i, x_i) \to (y_i, \pi_i)$ is an algorithm run by each solver, in which the solvers take an evaluation key ek, some external input $c_i \in \mathcal{X}$, and a possibly empty personal input $x_i \in \mathcal{X}$. Each solver then outputs $y_i \in \mathcal{Y}$ and a (possibly empty) proof π_i. The time taken for all solvers to run Eval, each using at most $O(poly(t, \lambda))$ processors, must be at least t.

$\mathsf{Verify}(\mathsf{vk}, X, Y, \Pi) \to \{(\mathsf{Yes}, \mathsf{aux}), (\mathsf{No}, \mathsf{aux})\}$ is an algorithm running in time $O(polylog(t), \lambda, n)$, which takes the verification key vk, along with a set of inputs $X = (\{c_i\}_{i \in R}, \{x_i\}_{i \in S})$, a set of outputs $Y = \{y_i\}_{i \in U}$, and a set of proofs $\Pi = \{\pi_i\}_{i \in V}$, where R, S, U, V are each a subset of the n solving parties. Verify then outputs Yes or No, along with some (possibly empty) auxiliary information aux.

On comparison of our definition with that of the single-party case given in [11], one can see that the latter is a special case of the former. The multi-party definition contains the following additional information: the number of solvers n, an optional personal input x_i, and some optional auxiliary information, aux. Let $n = 1$, and remove the optional values of x_i and aux. Upon observing that each set X, Y, Π will contain at most a single element in the single-party case, we have an equivalent definition to that of [11].

In proposing this new primitive, we differentiate between two classes of coVDFs, sequential and parallel, distinguished as follows.

Definition 2 (Sequential vs Parallel coVDFs). *A collaborative verifiable delay function is* sequential *if the external input of each solver (excluding the first) depends on the output of a previous solver. If all parties share a common, fixed external input, we instead say it is* parallel. *For a sequential construction we require that the first solver take some fixed external input. We refer to this, as well as the common external input in the parallel case, as the* seed c_0.

A coVDF, whether sequential or parallel, must satisfy correctness, soundness and sequentiality, as well as being unique and efficiently verifiable.

2.1 Correctness, Soundness and Sequentiality

We provide formal security definitions in the widely adopted framework of provable security, which allows for formal security proofs [35]. We start by defining correctness for a coVDF, which ensures that if the computation is performed correctly, then Verify will output Yes with overwhelming probability. We present the correctness game in Figure 1.

Figure 1 Correctness Game

$\mathbf{pp} = (\mathsf{ek}, \mathsf{vk}) \xleftarrow{R} \mathsf{Setup}(\lambda, t, n)$
`Setup is ran by the initiator`
for i in 1:n **do**
  $(y_i, \pi_i) \leftarrow \mathsf{Eval}(\mathsf{ek}, c_i, x_i)$
`All solving parties run the Eval algorithm`

$\{\mathsf{Yes}, \mathsf{No}\} \leftarrow \mathsf{Verify}(\mathsf{vk}, X = \{c_0, x_1, \cdots, x_n\}, Y = \{y_1, \cdots, y_n\}, \Pi = \{\pi_1, \cdots, \pi_n\})$

Definition 3 (Correctness). *A collaborative VDF is* correct *if for any* $\boldsymbol{pp}$ *there exists a negligible function in* λ, $\mathsf{negl}(\lambda)$ *such that the Correctness Game in Figure 1 outputs* Yes *with probability at least* $1-\mathsf{negl}(\lambda)$.

While correctness implies an honest evaluation overwhelmingly outputs Yes upon verification, the soundness property ensures that an incorrect evaluation of the protocol will not verify with overwhelming probability. We model this formally in Figure 2.

In the soundness game, the adversary, on input the public parameters, outputs a subset of parties M to subvert. They provide a personal input x_j and an external input c_j for each party in M. Our model is generic enough to capture both the sequential and parallel setting, where c_j is the output of solver $j-1$ or a fixed input, respectively. The adversary wins the game if Verify outputs Yes, for a false evaluation.

Definition 4 (Soundness). *A collaborative VDF is* sound *if for any PPT adversary* $\mathcal{A}$ *there exists a negligible function in* λ, $\mathsf{negl}(\lambda)$, *such that* $\mathcal{A}$ *cannot win the Soundness Game with probability greater than* $\mathsf{negl}(\lambda)$.

Figure 2 Soundness Game

$\mathbf{pp} \xleftarrow{\mathrm{R}} \mathsf{Setup}(\lambda, t, n)$
`Setup is ran by the initiator.`

$M \subseteq N \leftarrow \mathcal{A}(\mathbf{pp})$
`Adversary outputs a subset` M `of parties to subvert`

For $j \in M : (c_j, x_j) \leftarrow \mathcal{A}$
`Adversary outputs an external and personal input for each subverted party.`

For $j \in M : (y_j, \pi_j) \leftarrow \mathcal{A}(\mathbf{pp}, c_j, x_j)$
`Adversary produces output and proof.`

$\mathcal{A}$ wins if $\mathsf{Yes} \leftarrow \mathsf{Verify}(\mathsf{vk}, \{c_0, x_1, \cdots, x_n\}, \{y_1, \cdots, y_n\}, \{\pi_1, \cdots, \pi_n\}) \wedge\ (y_j, \pi_j) \neq \mathsf{Eval}(\mathsf{ek}, c_j, x_j)$, for any $j \in M$

Finally, we formalise the sequentiality property to capture how resistant a scheme is to parallelisation: an adversary with vast parallel resources should be able to evaluate the function no faster than αt, for some $\alpha \in (0, 1)$, close to 1.

In our game, shown in Figure 3, an adversary is given the public parameters for precomputation, before choosing an index j to subvert. The adversary produces a personal input, and an external input for party j. Using these, along with the preprocessing Z, the adversary attempts to replicate the correct output of Eval. $\mathcal{A}$ wins the game if this can be achieved, and security relies on precomputation and parallelisation providing a negligible speed up.

Definition 5 (Sequentiality). *Take any pair of randomised algorithms* $\mathcal{A}_0$ *and* $\mathcal{A}_1$, *where* $\mathcal{A}_0$ *runs in time* $O(poly(t, \lambda))$, *and* $\mathcal{A}_1$ *runs either in time* $\alpha(t)$ *(for a parallel construction) or* $\alpha(t/n)$ *(for a sequential construction). We call a coVDF* (p, α)-sequential *if for any adversary* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, *there exists a negligible function in* λ, $\mathsf{negl}(\lambda)$, *such that the Sequentiality Game cannot be won by* $\mathcal{A}$ *with probability greater than* $\mathsf{negl}(\lambda)$ *on at most* p *processors.*

Figure 3 Sequentiality Game

$\mathbf{pp} \xleftarrow{R} \mathsf{Setup}(\lambda, t, n)$
`Setup is ran by the initiator.`

$(Z, j, c_j, x_j) \xleftarrow{R} \mathcal{A}_0(\mathbf{pp})$
`Adversary preprocesses based on pp, and outputs an index` j`, an external input` c_j `and a personal input` x_j`.`

$(y_j, \pi_j) \leftarrow \mathcal{A}_1(Z, \mathbf{pp}, c_j, x_j)$
`Adversary produces output and proof.`

$\mathcal{A}$ wins if $(y_j, \pi_j) = \mathsf{Eval}(\mathsf{ek}, c_j, x_j)$.

Uniqueness and Efficient Verification. Additionally, a coVDF should satisfy uniqueness, and be efficiently verifiable. Similarly to a VDF [11], a coVDF satisfies uniqueness if the output of any one instance of a coVDF cannot be mangled into another, circumventing the delay. For a coVDF to have practical applications, a time gap is required between Eval and Verify. We define verification to be efficient if it runs in time $O(polylog(t), \lambda, n)$, where t is the time taken for all parties to run Eval. This is incorporated in Definition 1.

The properties we have presented above correspond to those required in a single-party VDF (cf. [11]). However, the presence of multiple parties introduces further security concerns, since some parties may not act honestly. As clock time is intrinsic to coVDFs and their applications, protocol aborts can be damaging. As such, we next define the additional properties of traceability and robustness, which help to prevent such aborts.

2.2 Security in a Trustless Setting

When used by a group of mutually trusting parties, the properties defined in Section 2.1 are sufficient for most applications. However, once we extend coVDFs to a trustless environment, e.g., for use in auctions (Section 1.2), we want to ensure that a malicious party can't simply abort the protocol, leading to the loss of computation. To achieve this, we introduce two new security properties for coVDFs: *traceability* and *robustness*.

Traceability. In order to provide incentives for good behaviour, we introduce *traceability*. The property of traceability has been used in a variety of primitives and protocols such as signature schemes and voting schemes [23, 25], and it typically involves the use of a trace algorithm, whose goal is to allow an entity (such as an administrator) to discover some information about some or all parties. In our context, this property is useful as it allows us to use the technique of *punishable abort* [6, 20, 7], where each party first places a deposit on a blockchain, and can only reclaim it if they act honestly.

To define traceability, we extend the definition of a coVDF to include a Trace algorithm. We define this algorithm in such a way that the tracer discovers all parties who acted dishonestly. The property of traceability is captured by defining the correctness and soundness of the trace algorithm, which ensure that no honest parties are output, and all malicious parties are output, respectively.

Trace is run after Verify outputs No in a run of the protocol. The tracer obtains the inputs, computation outputs and proofs of each party from Eval, and runs the Trace algorithm, which outputs a list of dishonest parties. Trace is formally described as follows.

$M \leftarrow \mathsf{Trace}(\mathsf{vk}, (\{c_i\}, \{x_j\}), \{y_k\}, \{\pi_h\})$, where $M \subseteq N$, is a PPT algorithm which takes the verification key vk, along with a set of inputs $(\{c_i\}, \{x_j\})$, a set of outputs $\{y_k\}$, and a set of proofs $\{\pi_h\}$, which are each a subset of size at most n. Trace outputs a list of misbehaving parties, corresponding to indices in M.

In Figure 4, we define the Trace *Correctness* Game where the trace algorithm is run on a coVDF instance where Verify outputs No. We say the tracer wins the Trace Correctness Game if no honest parties are output by Trace.

In Figure 5, we define the Trace *Soundness* Game we again run the trace algorithm on a coVDF instance where Verify outputs No. The adversary chooses a subset M of parties to subvert, and wins the Trace Soundness Game if for any party $j \in M$, their output is different from $\mathsf{Eval}(\mathsf{ek}, c_j, x_j)$, and yet j is not included in the set output by Trace.

Figure 4 Trace Correctness Game

$\mathbf{pp} \xleftarrow{\mathrm{R}} \mathsf{Setup}(\lambda, t, n)$
`Setup is ran by the initiator.`

Let X, Y, Π correspond to the inputs, outputs and proofs of all n solving parties.

$M \leftarrow \mathsf{Trace}(\mathsf{vk}, X, Y, \Pi)$
`Tracer` $\mathcal{C}$ `runs trace.`

$\mathcal{C}$ wins the game if $M \subseteq N \wedge \forall m \in M : \ (y_m, \pi_m) \neq \mathsf{Eval}(\mathsf{ek}, c_m, x_m)$

Definition 6 (Traceability). *A collaborative VDF satisfies* traceability *if for any public parameters, and any PPT adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, there exist negligible functions of λ such that the following hold*

1. *The tracer $\mathcal{C}$ wins the Trace Correctness Game with probability $1 - \mathsf{negl}_1(\lambda)$,*
2. *Adversary $\mathcal{A}$ cannot win the Trace Soundness Game with probability greater than $\mathsf{negl}_2(\lambda)$.*

Robustness. Many applications of coVDFs are time-sensitive, meaning the consequences of a protocol abort can be significant. We introduce *robustness* to capture the idea of a protocol being resistant to a fraction σ of malicious parties: if the number of dishonest parties is smaller than σn, the protocol can still run correctly, producing a time delay of t. We define robustness by letting the adversary control a fraction of the solvers, and causing Verify to fail on the set of honest users.

In Figure 6 we let an adversary $\mathcal{A}$ choose a subset of malicious parties $M \subseteq N$, and allow them to run any PPT algorithm. $\mathcal{A}$ wins the game if set M is smaller than σn and Verify run on the honest solvers is caused to output No.

Definition 7 (Robustness). *We say a collaborative VDF V is* σ-robust *for some $0 < \sigma < 1$ if for any PPT adversary $\mathcal{A}$ there exists a negligible function negl such that $\mathcal{A}$ wins the Robustness Game in Figure 6 with probability at most $\mathsf{negl}(\lambda)$.*

Figure 5 Trace Soundness Game

$\mathbf{pp} \xleftarrow{R} \mathsf{Setup}(\lambda, t, n)$
`Setup is ran by the initiator.`

$M \subseteq N \leftarrow \mathcal{A}(\mathbf{pp})$
`Adversary outputs a subset` M `of parties to subvert.`

For $j \in M : (x_j, c_j) \leftarrow \mathcal{A}$
`Adversary outputs a personal and external input for each subverted party.`

For $j \in M : (y_j, \pi_j) \xleftarrow{R} \mathcal{A}(\mathbf{pp}, c_j, x_j)$
`Adversary produces an output and proof for each subverted party.`

Let X, Y, Π correspond to the inputs, outputs and proofs of all n parties.

$Z \leftarrow \mathsf{Trace}(\mathsf{vk}, X, Y, \Pi)$
`Tracer` $\mathcal{C}$ `outputs set of cheating parties.`

$\mathcal{A}$ wins the game if $Z \subseteq N \wedge (y_j, \pi_j) \neq \mathsf{Eval}(\mathsf{ek}, c_j, x_j) \wedge j \notin Z$ for any $j \in M$

A scheme is robust if when the adversary corrupts a subset of the solvers, this corrupted subset can be identified and removed, with Verify outputting Yes on the subset of honest solvers. Note that this is distinguished from soundness, as $\mathcal{A}$ wins the soundness game (Figure 2) if Verify outputs Yes when run on the set of *all* outputs and proofs, rather than just the honest subset.

Figure 6 Robustness Game

$\mathbf{pp} \xleftarrow{R} \mathsf{Setup}(\lambda, t, n)$
`Setup is ran by the initiator.`

$M \subseteq N \leftarrow \mathcal{A}(\mathbf{pp})$
`Adversary outputs a subset` M `of parties to corrupt.`

For $k \in H = N \setminus M : (y_k, \pi_k) \leftarrow \mathsf{Eval}(\mathsf{ek}, c_k, x_k)$
`The honest parties run Eval.`

$\mathcal{A}$ wins if $|M| < \sigma n \wedge$
$(\mathsf{No}, \mathsf{aux}) \leftarrow \mathsf{Verify}\ (\mathsf{vk}, \{x_k\}_{k \in H}, \{y_k\}_{k \in H}, \{\pi_k\}_{k \in H})$

We note that our security model does not prevent an adversary from honestly computing multiple parts of a coVDF themselves. In doing so, however, they would expend significant effort, for no gain. Consider this in the applications given: In the collaborative work example, they would work on multiple parts, whereby a single part would be sufficient to provide access to some resource, meaning they have expended additional effort for no reason. Similarly, in an auction, this approach would mean making multiple bids, each requiring effort, where all but the largest bid is irrelevant. If a party wishes to compute the coVDF alone, then a standard VDF should be used instead.

We have now defined the properties of a coVDF. In the remainder of the paper we present our two candidate constructions for coVDFs: the first is a sequential construction which satisfies traceability but not robustness; this motivates our second, parallel construction, which is robust.

3 A sequential coVDF construction

In this section we take the VDF scheme presented by Wesolowski [38], and extend it to a coVDF. Our construction incorporates both personal inputs from each party as well as traceability.

In the construction given by Wesolowski in [38], the prover takes as input a base element x, performs a hash H on it, and then calculates $H(x)^{2^t}$, where t is the delay parameter. This calculation has to be done in a particular type of finite group, $\mathbb{G}$ - see [12] for a discussion on concrete groups. The verifier then uses a public-coin succinct argument [36] to efficiently verify the output. We will use this scheme to construct a coVDF in the following way.

Recall that H is a collision resistant hash function, let $\mathsf{Primes}(\lambda)$ be the set containing the first $2^{2\lambda}$ prime numbers, and let N be the set of n solvers. Let $\mathsf{bin}(x)$ be the representation of an element x as a binary string. We define $V_1 = (\mathsf{Setup}_1, \mathsf{Eval}_1, \mathsf{Verify}_1)$ as follows.

$\mathsf{Setup}_1(\lambda, t, n)$ takes security parameter λ, time delay t, and the number of solvers n and outputs public parameters $\mathbf{pp}$, which consist of a finite abelian group of unknown order $\mathbb{G}$, a hash function H mapping any string s to $\mathbb{G}$, and a hash function H_{prime} mapping any string s to $\mathsf{Primes}(\lambda)$. The public parameters also specify an ordering on the solvers.

The initiator randomly samples a seed $c_0 \in \mathbb{G}$, and passes it to the first solver. Each solver runs $\mathsf{Eval}_1(\mathbf{pp}, c_i, x_i)$ on an external input c_i and their personal input x_i, to compute y_i and π_i, as described in Figure 7. For the first solver, c_0 is used as the external input. For every other solver, $c_i \leftarrow y_{i-1}$ will be received from the previous solver.

Along with y_i, each solver also outputs three values which enable the trace algorithm to be non-interactive. These are z_i, τ_i and ω_i. τ_i and ω_i each consist of a tuple of the form (g^q, l), and are used to run Wesolowski's proof algorithm on the input x_i, and its inverse $z_i = x_i^{-1}$ respectively. These three values are only included for use in Trace, and are not used in Verify_1.

To verify the output computation, the verifier runs $\mathsf{Verify}_1(\mathbf{pp}, X, Y, \Pi)$ on the set of inputs, X, the set of outputs Y and the set of proofs Π. Solver n and the verifier run an extension of Wesolowski's succinct argument as described in Figure 9, and the verifier outputs either Yes or No.

Figure 7 Eval_1

1: Check $c_i \in \mathbb{G}$, and abort if not.
2: Compute $y_i \leftarrow x_i \cdot c_i^{2^t}$.
3: Pass y_i to solver $i+1$.
4: Compute the inverse of the personal input: $z_i \leftarrow x_i^{-1}$.
5: Compute $\pi_i \leftarrow z_i^{2^{(n-i)t}}$.
6: Compute τ_i and ω_i by running the proof algorithm in Figure 8 on $(c_i, y_i \cdot z_i)$ and (z_i, π_i) respectively.
7: Output (y_i^*, π_i^*), where $y_i^* = (y_i, z_i)$, and $\pi_i^* = (\pi_i, \tau_i, \omega_i)$.

We define the number of steps needed to be 2^{nt}, and split this into 2^t for each of the n solvers. During Eval_1, each player i computes 2^t modular exponentiations on c_i, and then multiplies this by x_i. The value $y_i \leftarrow x_i c_i^{2^t}$ is the output of player i, and the external input of player $i+1$. This value y_i will be raised to the power of $2^{(n-i)t}$ by the remaining solvers.

Figure 8 Proof algorithm

1: Take a pair of inputs (g, h).
2: Sample a prime $l \leftarrow H_{\mathsf{prime}}(\mathsf{bin}(g)||\mathsf{bin}(h))$.
3: The solver finds a linear combination of h and l, such that $h = ql + r$, with $0 \leq r \leq l$.
4: Output (g^q, l).

Figure 9 Verify_1

1: Solver n computes $y \leftarrow y_n \pi_1 \cdots \pi_n$.
2: The verifier checks $c_0, y \in \mathbb{G}$, and outputs No if not.
3: The verifier sends solver n a random prime $l \in \mathsf{Primes}(\lambda)$.
4: Solver n then finds a linear combination of 2^{nt} and l, such that $2^{nt} = ql + r$, with $0 \leq r \leq l$. $\tau := c_0^q$ is then sent to the verifier.
5: The verifier computes r from $2^{nt} \mod l$. If $\tau \in \mathbb{G}$ and $y = \tau^l c_0^r \in \mathbb{G}$, the verifier outputs Yes. If not, the verifier outputs No.

The output of player n can be written as

$$c_0^{2^{nt}} x_n x_{n-1}^{2^t} \cdots x_1^{2^{nt}}.$$

We want to cancel out all terms apart from $c_0^{2^{nt}}$. We require all parties to compute $z_i \leftarrow x_i^{-1}$ to achieve this. The personal input of player i will be raised to the power of $2^{(n-i)t}$ subsequent solvers. Therefore, solver i must then raise z_i to the power of $2^{(n-i)t}$. This allows us to 'unwrap' each of the inputs, by cancelling the personal input out with the correct power of z_i.

z_i is output along with y_i to enable traceability, which we describe later. Similarly, Figure 8 is run twice by each player i to compute proofs that both their output y_i, and their proof π_i are correct. This is to ensure a tracer has all the information needed to run the trace algorithm non-interactively, and the outputs τ_i and ω_i are not used in Verify_1.

In Verify_1, solver n multiplies their output y_n by each of the π_i values to obtain $y = c_0^{2^{nt}}$. This allows solver n and the verifier to run Wesolowski's succinct argument, which runs as follows.

The verifier first checks c_0 and y are in the group $\mathbb{G}$. The verifier then sends solver n a prime l taken from $\mathsf{Primes}(\lambda)$. Solver n writes 2^{nt} as a linear combination of l, such that $2^{nt} = ql + r$, with $r \leq l$. Solver n then computes $\tau \leftarrow c_0^q$ and sends this to the verifier. The verifier checks this $\tau \in \mathbb{G}$ and computes the value r from $2^{nt} \mod l$. The verifier then computes $\tau^l c_0^r$. If all parties have acted honestly, this is equal to $(c_0^q)^l c_0^r = c_0^{ql+r} = c_0^{2^{nt}} = y$.

Note that Verify_1 can be made non-interactive using the Fiat-Shamir heuristic [8] by sampling primes using H_{prime}, as in Figure 8.

Traceability. As discussed in Section 2.2, traceability allows one to remove dishonest parties prior to restarting the protocol. This allows a *punishable abort* [20, 7] to be incorporated into the scheme, providing an incentive to good behaviour.

In Eval_1, each party first checks their external input is in $\mathbb{G}$, and aborts the computation if it is not. If party i triggers such an abort, we immediately know party $i-1$ has acted dishonestly, removing the need for the trace algorithm.

In the case that all parties output $y_i, \pi_i \in \mathbb{G}$, but Verify_1 outputs No, we require a Trace algorithm. One way to obtain traceability is to run the succinct argument used in Verify_1 on each of the proofs and each of the outputs. Trace outputs all parties with an incorrect proof or output. In order to achieve this whilst ensuring that Trace is non-interactive, we have each solver compute

τ_i and ω_i to allow a tracer to run the succinct argument twice on each player to ensure they acted correctly.

We define $M \leftarrow \mathsf{Trace}_1(X, Y, \Pi)$ as an algorithm run by the tracer $\mathcal{C}$ on the set of all inputs (private and external), the set of all outputs and the set of all proofs, as prescribed in Figure 10. The output is the subset of dishonest parties, M.

Figure 10 Trace_1

1: Let M be an empty set.
2: **for** $i \in N$ **do**
3: Compute prime $l \leftarrow H_{\mathsf{prime}}(\mathsf{bin}(c_i)||\mathsf{bin}(y_i \cdot z_i))$.
4: Compute r_i from $y_i \cdot z_i \mod l$.
5: If $\tau_i \in \mathbb{G} \ \wedge \ y_i \cdot z_i = \tau_i^l c_i^{r_i} \in \mathbb{G}$ is not true, add i to M.
6: Compute prime $k \leftarrow H_{\mathsf{prime}}(\mathsf{bin}(z_i)||\mathsf{bin}(\pi_i))$.
7: Compute a_i from $\pi_i \mod k$.
8: If $\omega_i \in \mathbb{G} \ \wedge \ \pi_i = \omega_i^k z_i^{a_i} \in \mathbb{G}$ is not true, add i to M.
9: Output M.

The tracer runs Wesolowski's succinct argument, acting as the prover, twice for each party. On line 4, τ_i is used to prove that y_i was calculated correctly, and on line 6, ω_i is used to prove that π_i was calculated correctly. This is done using an identical argument to that in Verify_1. The primes are reconstructed to ensure they were correctly sampled from $\mathsf{Primes}(\lambda)$.

We have presented an efficient coVDF with the traceability property. This construction allows for applications such as collaborative work, as described in Section 1.2. In Appendix A we show concretely how this construction can be used in such a setting. In the following section, we provide a security analysis of this construction.

3.1 Security of V_1

In the full version of this paper, we provide a detailed security analysis of our parallel coVDF construction V_1, showing it satisfies the properties of correctness, robustness, sequentiality and uniqueness. Due to lack of space, here we instead provide a brief intuition for our results.

Correctness. On a correct run of the protocol, the personal input and proof will cancel out, meaning the underlying verification protocol will run on c_0 and $c_0^{2^{nt}}$, and hence output 1.

Soundness. Consider an adversary $\mathcal{A}$ who chooses a set of parties M to corrupt, along with inputs c_j, and x_j for each $j \in M$. This adversary then runs some algorithm $\mathcal{A}_1 \neq \mathsf{Eval}_1$ for each j. To break the soundness property, they would have to output a pair (y_j, π_j) such that $c_j^{2^t}(x_j^{2^{n-j}}\pi_j) = y_j \neq 1_{\mathbb{G}}$ for each j. Even if $\mathcal{A}$ lets $x_j = \pi_j = 1$, then they still have to find a solution for t sequential squarings of c_j without running the squaring algorithm. This reduces to the adaptive root assumption of the underlying construction [38], and so has an overwhelmingly small probability of being correct.

Sequentiality. The sequentiality of this scheme directly relies upon the task of repeated squaring. This has been a base assumption for many time-lock constructions [34, 28], and is considered a standard assumption.

Traceability. Trace correctness holds if all parties output by Trace have not run $\mathsf{Eval}_1(\mathbf{pp}, c_i, x_i)$. Any party who runs $\mathsf{Eval}_1(\mathbf{pp}, c_i, x_i)$ will have the correct values of τ_i and ω_i, as well as the correct values of z_i, π_i and y_i. This gives us trace correctness. Meanwhile, trace soundness holds if all parties j who output $(y_j^*, \pi_j^*) \neq \mathsf{Eval}_1(\mathbf{pp}, c_i, x_i)$ are output by Trace. Trace runs Wesolowski's succinct

argument twice: once on the output y_j, and once on the proof π_j. As this procedure is deterministic, any party j who outputs $(y_j^*, \pi_j^*) \neq \mathsf{Eval}_1(\mathbf{pp}, c_i, x_i)$ will fail one of the succinct arguments, and so be output by Trace.

4 A robust coVDF construction

In this section, we propose a parallel coVDF construction based on repeated hashing, which we will see achieves 1/3-robustness, i.e., it withstands up to $n/3$ malicious parties whilst still running correctly. Such a parallel scheme will be particularly useful in applications such as blind auctions, as discussed in Section 1.2 and discussed more in detail in Appendix A.

Sequential hashes have been used in a similar scenario: in [29], Mahmoody et al. provide a proof of sequential work based on a directed acyclic graph (DAG), in which a prover first creates a DAG, and then hashes all edges between the nodes of the graph. This creates a tree of hashes, which can be verified probabilistically by checking some fraction of the hashes. This construction naturally satisfies most of the properties of a verifiable delay function, however it does not satisfy uniqueness. This is because for a given solution, changing a single edge will provide a different output, whilst being unlikely to be picked up by random challenges.

In this section, we look at a method of mitigating this whilst extending the scheme to multiple parties. Rather than creating a suitable DAG prior to the hashing procedure, we instead create it during the hashing phase at random. This is achieved by solvers randomly syncing to another party, who then creates a hash to mark the event. This process stops once some parties have completed t hashes, achieving the required delay.

To ensure each party generates the same graph, and to enable us to prove security notions, we base our construction on the hashgraph consensus protocol proposed in [3].

The hashgraph consensus protocol. The Swirlds hashgraph consensus protocol [3] is an asynchronous Byzantine Fault Tolerance consensus protocol which is proved to be fast and fair. Here we give a brief description of the key concepts of this protocol, and in the full version of this paper, we present the relevant definitions and results from [3] more formally. We refer the reader to [3] for a full description and proofs of the ideas presented.

Gossip about gossip. The underlying idea of this protocol is that as often as possible each party runs a gossip protocol[2] in which they sync with another party, transmitting all the new events ('gossip') they have learned from previous syncs from other parties. Using this method, all information will spread exponentially fast through the group of parties. Each time a sync occurs, the party receiving the sync creates a new node, also known as an event, by hashing their most recent event concatenated with the most recent event of the party who initiated the sync. Through repeated syncing, each party will eventually end up with a copy of the same graph, up to a certain point in time.

Sequential hashing makes this scheme resistant to parallelisation, and the required delay is defined by the number of hashes computed by each solver.

[2] A gossip protocol is a procedure in which nodes propagate information through a group, based on the way epidemics spread; see [18, 26].

4.1 coVDF from repeated hashing

In this construction, parties build a graph together, populating it with events, or nodes. It is necessary that all parties reach consensus on the ordering of nodes. This ensures that each party has a copy of the same graph, allowing for a single output which can be efficiently verified. To achieve a unique output, and to prove robustness, our protocol uses features from [3]. In this section we outline the major intuitive ideas, presenting the technical details in the security analysis provided in the full version of this paper.

We first write our coVDF as a triple of algorithms: $V_2 = (\mathsf{Setup}_2, \mathsf{Eval}_2, \mathsf{Verify}_2)$ as follows.

Setup. $\mathsf{Setup}_2(\lambda, t, n) \rightarrow \mathbf{pp} = (H, \mathcal{G}, m)$ takes a security parameter λ, time delay t, and the number of solvers n; and outputs public parameters $\mathbf{pp}$=$(H, \mathcal{G}, m)$, where H is a collision resistant hash function H: $\{0,1\}^{2m} \rightarrow \{0,1\}^m$, $\mathcal{G}$ is an empty directed acyclic graph, containing a random ordering of the n parties dictating where each party's events will sit on the graph, and m is used to define the hash function and the input space, $\mathcal{X} = \{0,1\}^m$. The output space is defined to be a populated graph of depth at least t. Additionally, we require each personal input x_i to be some data hashed by H.

Eval. The initiator randomly samples an external input $c_0 \in \{0,1\}^m$, and passes it to the solvers. Each of the n solvers runs $\mathsf{Eval}_2(\mathbf{pp}, c_0, x_i) \rightarrow (\mathcal{G}, \mathcal{H})$ by taking the external input c_0, providing a personal input x_i, and doing as prescribed in Figure 11. The output is $\mathcal{G}$, the completed directed graph up to the point where consensus has been reached, along with $\mathcal{H}$, the set of parties who agree with the output graph $\mathcal{G}$, voting Yes on Line 9 of Figure 11.

Figure 11 Eval_2

```
1: Compute i's first event H(x_i||c_0).
2: Let G be the graph output by Setup_2.
3: while At least n/3 different parties have fewer than t nodes do
4:     Call Sync as both sender and receiver in parallel, with different parties.
5: for j in 1 : n/3 do
6:     if i = j then
7:         Output graph G_j up to consensus.
8:     else
9:         Vote Yes if graphs G_i and G_j are consistent, and No otherwise.
10:    If graph gets at least 2n/3 votes, end Eval and output this graph.
11: Abort protocol.
```

In the Eval_2 algorithm, each party first hashes their personal input with c_0, before repeatedly running a gossip protocol (Figure 12). Each party then repeatedly syncs at random with other parties, creating a hash to record each sync. During each sync, the syncing party, i, will tell the receiving party, j, all events that have been gossiped to i since the last sync between i and j. This sync is known as gossip.

Figure 12 Sync

1: Let i be the sender, and let j be the receiver.
2: i sends j all new events since their previous sync.
3: j updates $\mathcal{G}$ with these new events, and creates a new event $H(x||y)$, where x and y are the most recent events by j and i respectively.
4: j calls divideRounds
5: j calls decideFame
6: j calls findOrder
7: Output $\mathcal{G}_j$

This process will stop once at least $2n/3$ parties have computed t hashes (see Line 3). Any party who has calculated less than ϕt hashes will be dropped from the group prior to verification, where ϕ is a parameter representing the minimum amount of work required to not be considered lazy.

Next, parties vote on which graph to output. This is achieved as follows: Party 1 outputs their graph up to $(1-\epsilon)t$ nodes (this is the point at which consensus has been reached on the graph - we go into greater detail on ϵ later). All parties check their copy of the graph is consistent with that of party 1. They then vote Yes or No accordingly. If at least $2n/3$ votes are Yes, this graph is output. If not, party 2 outputs their graph and repeats the process. This can be repeated up to $n/3$ times. If no graph receives enough votes by this point, the protocol is aborted.

Note that the last three lines of Figure 12 are commands to run three new procedures, all of which can be found in the full version of this paper, as well as in [3]. These procedures are necessary to provide uniqueness and robustness, but not to provide an intuition of the scheme. In short, these three algorithms ensure that all parties have the same ordering on all events, by splitting the graph up into epochs, and marking the first node in each epoch as a witness. Witnesses which are quickly gossiped to more than $2n/3$ of parties are then called famous. divideRounds is used to determine the round of events in the previous round, decideFame is used to determine whether a witnesses in previous rounds are famous, findOrder is used to provide consensus on the ordering of events. The notions of a node being a witness, famous and having a round number are expanded on in the full version of this paper, as well as in [3].

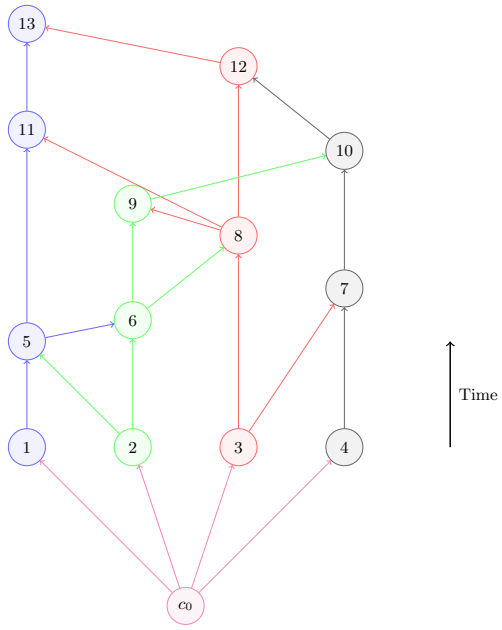
We now provide an example to illustrate how the graph is built using Eval_2, before moving on to Verify_2 and the security analysis.

Each node is calculated from the hash of two previous nodes; the most recent node by the receiver of the sync, concatenated with the most recent node of the initiator of the sync. We refer to these two nodes as the *parents* of the new node. We will use the notation $k||l$ to refer to the hashing of node k concatenated with node l. In Figure 13, the two ingoing arrows to each node represent the parents.

In Figure 13, we show the start of a graph with $n = 4$ being populated. Each party's first event, $\{1, 2, 3, 4\}$, is the result of hashing $x_i||c_0$. After this, each party repeatedly syncs with other parties, and the receiving party creates a new node each time. For example node 5 is created by hashing $1||2$, and node 8 is created by hashing $3||6$.

We give an example of the next sync to occur after Figure 13, leading to a new event (which would be node 14) being added to the graph: solver 1 (Blue) syncs with solver 2 (Green). We see that Blue knows every event, as each party's most recent event has been gossipped via $9 \to 10 \to 12 \to 13$.

Figure 13 Example



In the most recent sync between Blue and Green, Green had events $\{1, 2, 3, 4, 5, 6\}$ in their graph. Therefore Blue sends the hash labels and corresponding parents for nodes $\{7, \cdots, 13\}$. Green now updates their graph and adds the new node 14 by hashing $9||13$.

Now, if Red were to sync with Green, Red would also send events $\{4, 7, 10, 12\}$ as these are the events Red has learned since their previous sync with Green (which resulted in the creation of node 9). Green would check that these were consistent with those currently in their graph, and add event 15 by hashing $14||12$.

Verify. The verify algorithm $\mathsf{Verify}_2(H, c_0, \mathcal{G}, \mathcal{H}) \rightarrow \{\mathsf{Yes}, \mathsf{No}\}$ takes the hash function H, the external input c_0 and the graph $\mathcal{G}$, along with the subset $\mathcal{H}$ who voted Yes on the graph.

Figure 14 Verify_2

On graph $\mathcal{G}$, choose $k(t)$ nodes created by each party in $\mathcal{H}$. These are the challenge nodes.
for All challenge nodes **do**
    Hash the two parents of each node together and compare with the node. If they are different, remove the node creator from $\mathcal{H}$.
if $\#\mathcal{H} > 2n/3$ **then**
    Output Yes.
else
    Output No.

The verifier then checks $k(t) = \omega(\lambda) \log t$ hashes at random for each player, where $\omega(\lambda)$ is an increasing function which ensures the probability of any malicious party remaining in $\mathcal{H}$ is negligible with respect to the security parameter. Any party found to have incorrectly computed a hash will be removed from $\mathcal{H}$. If the number of parties remaining in $\mathcal{H}$ is greater than $2n/3$ after all checks

are completed, the verifier outputs Yes.

Efficiency remarks. This construction is based upon a fast, fair consensus protocol which makes very efficient use of bandwidth.

Setup_2 simply provides an ordering of the parties, as well as specifying a cryptographic hash function, and the size of inputs. This requires little computational overhead.

Eval_2 repeatedly calls a subroutine Sync (Figure 12), which involves one party i sending j the set of new events i learned since their precious sync.

Importantly, sync is run in parallel, meaning all parties can act as both the receiver and the sender simultaneously. As soon as party j has completed the hash, and added the new events from the sync, they can receive another sync whilst running the procedures divideRounds, decideFame and findOrder. This ensures each party is continuously hashing new events.

In [3] it is shown that the gossip protocol is very efficient in terms of bandwidth: each member will receive each transaction once, and also send each transaction on average once. The hashes for gossiped events don't have to be sent during the sync - it is sufficient to send the identity of the creator of the event, and the event number of its other parent, which can be stored in a single array.

V_2 is a realistic construction, with the potential to be used in practice. However, it should be noted that the verification time grows linearly in the number of parties, making this less suited to large values of n.

4.2 Security of V_2

In the full version of this paper, we provide a detailed security analysis of our parallel coVDF construction V_2, showing it satisfies the properties of correctness, robustness, sequentiality and uniqueness. Due to lack of space, here we provide a brief intuition.

Correctness. If all parties run the Eval_2 algorithm, then they will construct a graph with a minimum depth of t, and all nodes will have been hashed correctly. Hence all checks will be correct, and Verify_2 will output Yes.

Robustness. For robustness to hold we require that provided at least $2n/3$ parties act honestly, the computation should be accepted by the verifier, regardless of the behaviour of the remaining parties. We analyse the possible behaviour of malicious parties, which include lack of participation, faking hash values, and dishonest gossiping. We utilise results from Baird [3] to show that in each case, we still get consistency under an honest majority of 2/3.

Sequentiality. The sequentiality of this construction is based upon sequential hashing, which has been used previously as a proof of sequential work [29]. The sequentiality of V_2 follows directly from the assumption that iterated hashing is resistant to parallelisation.

Uniqueness We show that assuming that we have a 2/3 honest majority of parties, then an output graph cannot be mangled into another such graph. This is due to the graph relying directly on the set of personal inputs, which due to the collision resistance of the hash function cannot be recovered.

5 Concluding remarks

In this work we introduced the idea of collaborative VDFs, extending the definitions of the single party case. We formalized the primitive and its security properties, and we categorised coVDFs

as either sequential or parallel. We additionally defined the new properties of traceability and robustness to address the behaviour of dishonest solvers in the multi-party setting.

An approach to achieving a sequential construction is to take an existing VDF scheme and splitting the work into n parts, where each solver computes one such part before passing the work on to the next solver. We proved this is possible by giving a concrete extension of Wesolowski's VDF [38], additionally providing traceability. We then proposed a candidate parallel construction in which all parties build a graph together, using gossip and sequential hashing. These constructions show the flexibility of this primitive, and allow the use of coVDFs in a variety of applications.

References

1. Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *Theory of Cryptography Conference*, pages 137–156. Springer, 2007.
2. Samiran Bag, Feng Hao, Siamak F. Shahandashti, and Indranil G. Ray. Seal: Sealed-bid auction without auctioneers. *IEEE Transactions on Information Forensics and Security*, 2019.
3. Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. 2016.
4. Leemon Baird, Mance Harmon, and Paul Madsen. Hedera: A governing council & public hashgraph network. 2018.
5. G. Bakos. Energy management method for auxiliary energy saving in a passive-solar-heated residence using low-cost off-peak electricity. *Energy and Buildings*, 31(3):237 – 241, 2000.
6. Carsten Baum, Rafael Dowsley, and Bernardo David. Insured mpc: Efficient secure multiparty computation with punishable abort. *Financial Cryptography and Data Security 2020*, 2020.
7. Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 421–439. Springer Berlin Heidelberg, 2014.
8. David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. In *Advances in Cryptology - ASIACRYPT 2012*, pages 626–643. Springer, 2012.
9. Ken Birman. The promise, and limitations, of gossip protocols. *SIGOPS Operating Systems Review*, page 8–13, 2007.
10. Erik-Oliver Blass and Florian Kerschbaum. Borealis: Building block for sealed bid auctions on blockchains. Cryptology ePrint Archive, Report 2019/276, 2019. https://eprint.iacr.org/2019/276.
11. Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 757–788. Springer, 2018.
12. Dan Boneh, Benedikt Bünz, and Ben Fisch. A survey of two verifiable delay functions. IACR Cryptology ePrint Archive, 2018. https://eprint.iacr.org/2018/712.pdf.
13. Bram Cohen and Krzysztof Pietrzak. Simple proofs of sequential work. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 451–467, Cham, 2018. Springer International Publishing.
14. Bram Cohen and Krzysztof Pietrzak. The chia network blockchain, 2019.
15. Vicki M Coppinger, Vernon L Smith, and Jon A Titus. Incentives and behavior in english, dutch and sealed-bid auctions. *Economic Inquiry*, pages 1–22, 1980.
16. Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. Continuous verifiable delay functions. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 125–154. Springer, 2020.
17. Luca De Feo, Simon Masson, Christophe Petit, and Antonio Sanso. Verifiable delay functions from supersingular isogenies and pairings. *Advances in Cryptology – ASIACRYPT 2019*, pages 248–277, 2019.
18. Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Boris Koldehofe, Martin Mogensen, Maxime Monod, and Vivien Quéma. Heterogeneous gossip. In Jean M. Bacon and Brian F. Cooper, editors, *Middleware 2009*, pages 42–61. Springer, 2009.
19. Hisham S Galal and Amr M Youssef. Verifiable sealed-bid auction on the ethereum blockchain. In *International Conference on Financial Cryptography and Data Security*, pages 265–278. Springer, 2018.
20. Hongmin Gao, Zhaofeng Ma, Shoushan Luo, and Zhen Wang. Bfr-mpc: A blockchain-based fair and robust multi-party computation scheme. *IEEE Access*, pages 110439–110450, 2019.

21. Ronald Huisman, Christian Huurman, and Ronald Mahieu. Hourly electricity prices in day-ahead markets. *Energy Economics*, 29(2):240 – 248, 2007.
22. Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 357–388, Cham, 2017. Springer International Publishing.
23. Aggelos Kiayias, Yiannis Tsiounis, and Moti Yung. Traceable signatures. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 571–589. Springer Berlin Heidelberg, 2004.
24. Esteban Landerrechel, Marc Stevens, and Christian Schaffner. Non-interactive cryptographic timestamping based on verifiable delay functions. *FC 2020: Financial Cryptography and Data Security 2020*, 2020.
25. Lezhen Ling and Junguo Liao. Anonymous electronic voting protocol with traceability. In *2011 International Conference for Internet Technology and Secured Transactions*, pages 59–66, 2011.
26. Harry C. Li, Allen Clement, Edmund L. Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. Bar gossip. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, page 191–204. USENIX Association, 2006.
27. Anna Lysyanskaya and Nikos Triandopoulos. Rationality and adversarial behavior in multi-party computation. In *Annual International Cryptology Conference*, pages 180–197. Springer, 2006.
28. Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Time-lock puzzles in the random oracle model. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 39–50. Springer, 2011.
29. Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Publicly verifiable proofs of sequential work. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*, page 373–388. Association for Computing Machinery, 2013.
30. Preston McAfee and John McMillan. Auctions with a stochastic number of bidders. *Journal of economic theory*, 43:1–19, 1987.
31. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
32. Krzysztof Pietrzak. Simple verifiable delay functions. *10th Innovations in Theoretical Computer Science Conference (ITCS 2019)*, 2019.
33. M. S. Riazi, M. Javaheripi, S. U. Hussain, and F. Koushanfar. Mpcircuits: Optimized circuit generation for secure multi-party computation. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 198–207, 2019.
34. Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.
35. Phillip Rogaway and Yusi Zhang. Simplifying game-based definitions. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 3–32, Cham, 2018. Springer International Publishing.
36. Nigel Smart. *Cryptography: an introduction*, volume 3. McGraw-Hill New York, 2003.
37. Jon Truby. Decarbonizing bitcoin: Law and policy choices for reducing the energy consumption of blockchain technologies and digital currencies. *Energy research & social science*, 44:399–410, 2018.
38. Benjamin Wesolowski. Efficient verifiable delay functions. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 379–407. Springer, 2019.

A Examples

We provide two concrete examples illustrating how to use each of our proposed coVDF constructions in the applications discussed in Section 1.2.

A.1 Collaborative Work

We consider a scenario where a group of n mutually distrusting parties wish to access a private resource owned by a company. In return for this access, the company asks the group to compute a coVDF in order to implement a randomness beacon.

The company wishes to ensure that only parties who contributed gain access, and parties all want to contribute the same amount of effort, whilst keeping their identity hidden from other parties. We assume that each party has a private communication channel with the company, and

that each party deposits some funds on a blockchain, under the condition that this is returned only if they carry out the protocol honestly.

Let the required length of evaluation be nt, and let each party have some personal input $x_i \in \mathbb{G}$, obtained by hashing their public identifier. By setting these hashes as the personal inputs, we show how this group can together compute such a coVDF.

The company runs $\mathsf{Setup}_1(\lambda, t, n) \rightarrow \mathbf{pp} = (\mathbb{G}, H, H_{\mathsf{prime}})$, taking as input a chosen security parameter λ and time delay t, and the number of parties n. The outputs are the public parameters, which consist of a group $\mathbb{G}$, as well as hash functions H and H_{prime}, as described in Section 3. The company then chooses some ordering on the parties, and provides the starting party with a seed c_0, sampled at random from $\mathbb{G}$.

Parties run Eval_1 in turns, as described in Section 3, using their hashed identifiers x_i as personal inputs. The first party will use c_0 as their external input, and subsequent parties will use the output of the previous party, $c_i = y_{i-1}$. Then the final party and the company run Verify_1, which will output Yes or No.

If Verify outputs Yes, we can use Trace to allow parties to each privately reveal their identity to the company. This is done by running lines 6 to 8 of Trace on each party, to verify that x_i, was indeed the personal input of i, and then the party reveals their public identifier. After checking that this identifier hashes to x_i, The company will then give i access to the resource, and each party will receive their deposit back.

If Verify outputs No, Trace is ran by the company, and honest parties are refunded their deposit. The remaining funds can be split between the authority and honest parties.

A.2 Decentralised Blind Auctions

In our second example, we show how to instantiate a fair, decentralised blind auction using a parallel coVDF. Consider a seller who wishes to auction off some goods in a decentralised fashion, to avoid the fees associated with an auction house. Suppose this seller requires interested parties to show legitimate interest with a proof of effort (i.e. a delay of time t), to avoid fake bids. We allow the n interested parties place bids in this auction, as discussed in Section 1.2.

We can achieve this by using our proposed parallel coVDF, V_2. Parties all hash their bids concatenated with some randomness at the same time, and each compute t steps of iterated computation, serving as the proof of effort. This proceeds as follows:

Once the seller has advertised this auction and is ready to accept bids, they will run Setup_2 on the number of interested parties n. This means that when all parties bid, they have full transparency of the number of parties involved in this auction, providing a fair setting.

$\mathsf{Setup}_2(\lambda, t, n) \rightarrow \mathbf{pp} = (H, \mathcal{G}, m)$ takes a security parameter λ, time delay t, and the number of solvers n; and outputs public parameters $\mathbf{pp}$=$(H, \mathcal{G}, m)$, which give the parties the necessary details to produce a hashgraph. Parties can now hash their bid using H to obtain their personal input x_i. The seller then randomly samples an element $c_0 \in \mathcal{G}$ as the external input allowing parties to use this c_0 and their personal inputs x_i to run Eval_2, together producing a graph of depth t.

We have now successfully instantiated an auction, and we can use standard techniques from secure multi party computation to calculate the highest bid, whilst keeping all others private. An example of this is Borealis [10], which is an efficient, low interaction protocol for secure computation of rank among integers.