Universally Verifiable Multiparty Computation from Threshold Homomorphic Cryptosystems

Berry Schoenmakers and Meilof Veeningen, Eindhoven University of Technology

{berry@win.,m.veeningen@}tue.nl

Abstract. Multiparty computation can be used for privacy-friendly outsourcing of computations on private inputs of multiple parties. A computation is outsourced to several computation parties; if not too many are corrupted (e.g., no more than half), then they cannot determine the inputs or produce an incorrect output. However, in many cases, these guarantees are not enough: we need correctness even if *all* computation parties may be corrupted; and we need that correctness can be verified even by parties that did not participate in the computation. Protocols satisfying these additional properties are called "universally verifiable". In this paper, we propose a new security model for universally verifiable multiparty computation, and we present a practical construction, based on a threshold homomorphic cryptosystem. We also develop a multiparty protocol for jointly producing non-interactive zero-knowledge proofs, which may be of independent interest.

Keywords: multiparty computation, verifiability, Fiat-Shamir heuristic, threshold homomorphic cryptosystem

1 Introduction

Multiparty computation (MPC) provides techniques for privacy-friendly outsourcing of computations. Intuitively, MPC aims to provide a cryptographic "black box" which receives private inputs from multiple "input parties"; performs a computation on these inputs; and provides the result to a "result party" (an input party, a particular third party, or the public). This black box is implemented by distributing the computation between multiple "computation parties", with privacy and correctness being guaranteed in case of passive corruptions (e.g., [BCD$^+$09]), active corruption of a minority of computation parties (e.g., [CDN01]), or active corruption of all-but-one computation parties (e.g., [DPSZ12]).

However, multiparty computation typically does *not* provide any guarantees in case all computation parties are corrupted. That is, the result party has to trust that at least some of the computation parties did their job, and has no way of independently verifying the result. In particular, the result party has no way of proving to an external party that his computation result is indeed correct. *Universally verifiable* multiparty computation addresses these issues by requiring that the correctness of the result can be verified by any party, even if all computation parties are corrupt [dH12]; it was originally introduced in the context of e-voting [CF85,SK95], but is also relevant for other applications of MPC where external parties rely on the results of a computation [BCD$^+$09,dHSCodA14].

Unfortunately, the state-of-the-art on universally verifiable MPC is unsatisfactory. The concept of universally verifiable MPC was first proposed in [dH12], where it was also suggested that it can be achieved for MPC based on threshold homomorphic cryptosystems. However, [dH12] does not provide a rigorous security model for universal verifiability or analysis of the proposed construction; and the construction has some technical disadvantages (e.g., a proof size depending on the number of computation parties). The scheme recently proposed in [BDO14] solves part of the problem. Their protocols provide "public auditability", meaning that anybody can verify the result of a computation, but *only* if that result is public. In particular, it is not possible for a result party to prove just that an encryption of the result is correct, which is important if this result is to be used in a later protocol without being revealed.

In this paper, we propose a new security model for universally verifiable multiparty computation, and a practical construction achieving it. As in [dH12], we adapt the well-known actively secure MPC protocols based on threshold homomorphic cryptosystems from [CDN01,DN03]. Essentially, these protocols perform computations on encrypted values; security against active adversaries is achieved by letting parties prove correctness of their actions using interactive zero-knowledge proofs. Such interactive proofs only convince parties present at the computation; but making them non-interactive makes them convincing also to external parties. Concretely, the result of a computation is a set of encryptions of the inputs, intermediate values, and outputs of the computation, along with non-interactive zero-knowledge proofs of their correctness. We improve on [dH12] by providing a security model for universal verifiability (in the random oracle model), and security proofs for our protocols; by eliminating the need for trapdoor commitments; and by making the proof size independent of the number of parties performing the computation. We achieve the latter using a new "multiparty" variant of the Fiat-Shamir heuristic that homomorphically combines contributions from the different parties, which may be of independent interest.

As such, universally verifiable MPC provides a practical alternative to recent (single-party) techniques for verifiable outsourcing. Specifically, many papers on verifiable computation focus on efficient verification, but do not cover privacy [PHGR13,WB13]. Those works that do provide privacy, achieve this by combining costly primitives, e.g., fully homomorphic encryption with verifiable computation [FGP14]; or functional encryption with garbled circuits [GKP$^+$13]. A recent work [ACG$^+$14] also considers the possibility of achieving verifiable computation with privacy by distributing the computation; but it does not guarantee correctness if all computation parties are corrupted, nor does it allow third parties to be convinced of this fact. In contrast, our methods guarantee correctness even if all computation parties are corrupted, and even convince other parties than the input party. In particular, any third party can be convinced, and the computation may involve the inputs of multiple mutually distrusting input

$a \in_R S$	Sample a uniformly at random from S
$\mathsf{recv}(\mathcal{P}), \mathsf{send}(v, \mathcal{P})$	Send/receive v to/from $\mathcal{P}$ over secure channel
$\mathsf{bcast}(v)$	Exchange v over broadcast channel
party $\mathcal{P}$ **do** S	Let party $\mathcal{P}$ perform S; other parties do nothing
foreach party $i \in \mathcal{Q}$ **do** S	Let parties $i \in \mathcal{Q}$ perform S in parallel
$\mathcal{H} : \{0,1\}^* \to \{0,1\}^{2l}$	Cryptographic hash function (l security parameter)
$F \subset \mathcal{I} \cup \mathcal{P} \cup \{\mathcal{R}, \mathcal{V}\}$	Global variable: set of parties found to misbehave
$\mathsf{paillierdecode}(x)$	Threshold Paillier decoding (p. 6): $((x-1) \div N)(4\Delta^2)^{-1} \bmod N$
$\mathsf{fsver}_\Sigma(v, a, c, r, aux)$	Verification of Fiat-Shamir Σ-proof (p. 7): $\mathcal{H}(v\|\|a\|\|aux) = c \wedge \Sigma.\mathsf{ver}(v, a, c, r)$

Fig. 1. Notation in algorithms, protocols, and interactive Turing machines

parties. Moreover, in contrast to the above works, our methods rely on basic cryptographic primitives such as Σ-protocols and the threshold homomorphic Paillier cryptosystem, readily available nowadays in cryptographic libraries like SCAPI [EFLL12].

Outline First, we briefly recap the CDN scheme for secure computation in the presence of active adversaries from [CDN01,DN03], instantiated using Paillier encryption (Section 2). Then, we show how the proofs in this protocol can be made non-interactive using the Fiat-Shamir heuristic and our new multiparty variant (Section 3). Finally, we propose a security model for universally verifiable MPC, and show that CDN with non-interactive proofs is universally verifiable (Section 4). We conclude in Section 5. We list potentially non-obvious notation in our pseudocode in Figure 1.

2 Secure Computation from Threshold Cryptography

We review the "CDN protocol" [CDN01] for secure computation in the presence of active adversaries based on a threshold homomorphic cryptosystem. The protocol involves m input parties $i \in \mathcal{I}$, n computation parties $i \in \mathcal{P}$, and a result party $\mathcal{R}$. The aim of the protocol is to compute a function $f(x_1, \ldots, x_m)$ (seen as an arithmetic circuit) on private inputs x_i of the input parties, such that the result party obtains the result.

2.1 Computation using a Threshold Homomorphic Cryptosystem

The protocol uses a (t, n)-threshold homomorphic cryptosystem, with $t = \lceil n/2 \rceil$. In such a cryptosystem, anybody can encrypt a plaintext using the public key; add two ciphertexts to obtain a (uniquely determined) encryption of the sum of the corresponding plaintexts; and multiply a ciphertext by a constant to obtain a (uniquely determined) encryption of the product of the plaintext with the constant. Decryption is only possible if at least t out of the n decryption keys are known. A well-known homomorphic cryptosystem is the Paillier cryptosystem [Pai99]: here, the public key is an RSA modulus $N = pq$; $a \in \mathbb{Z}_N$

is encrypted with randomness $r \in \mathbb{Z}_N^*$ as $(1+N)^a r^N$; and the product of two ciphertexts is an encryption of the sum of the two corresponding plaintexts. (In this paper, we suppress moduli for readability.) A threshold variant of this cryptosystem was presented in [DJ01]. The (threshold) decryption procedure is a bit involved; we postpone its discussion until Section 2.2. The CDN protocol can also be instantiated with other cryptosystems; but in this paper, we will focus on the Paillier instantiation.

Computation of $f(x_1, \ldots, x_m)$ is performed in three phases: the input phase, the computation phase, and the output phase. In the input phase, each input party encrypts its input x_i, and broadcasts the encryption X_i. In the computation phase, the function f is evaluated gate-by-gate. Addition and subtraction are performed using the homomorphic property of the encryption scheme. For multiplication[1] of X and Y, each computation party $i \in \mathcal{P}$ chooses a random value d_i, and broadcasts encryptions D_i of d_i and E_i of $d_i \cdot y$. The computation parties then compute $X \cdot D_1 \cdots D_n$, and threshold decrypt it to learn $x + d_1 + \ldots + d_n$. Observe that this allows them to compute an encryption of $(x+d_1+\ldots+d_n)\cdot y$, and hence, using the E_i, also an encryption of $x \cdot y$. Finally, in the output phase, when the result of the computation has been computed as encryption X of x, the result party obtains x by broadcasting random encryption D of d and obtaining a threshold decryption $x - d$ of $X \cdot D^{-1}$.

Active security is achieved by letting the parties prove correctness of all information they exchange. Namely, the input parties prove knowledge of their inputs X_i (this prevents parties from choosing inputs depending on other inputs). The computation parties prove knowledge of D_i, and prove that E_i is indeed a correct multiplication of D_i and Y; and they prove the correctness of their contributions to the threshold decryption of $X \cdot D_1 \cdots D_n$ and $X \cdot D^{-1}$. Finally, the result party proves knowledge of D. We now discuss these proofs of correctness and their influence on the security of the overall protocol.

2.2 Proving Correctness of Results

The techniques in the CDN protocol for proving correctness are based on Σ-protocols. Recall that a Σ-protocol for a binary relation R is a three-move protocol in which a potentially malicious prover convinces a honest verifier that he knows a *witness* w for *statement* v such that $(v, w) \in R$. First, the prover sends an *announcement* (computed using algorithm $\Sigma.\mathsf{ann}$) to the verifier; the verifier responds with a uniformly random *challenge*; and the prover sends his *response* (computed using algorithm $\Sigma.\mathsf{res}$), which the verifier verifies (using predicate $\Sigma.\mathsf{ver}$). Σ-protocols satisfy the following properties:

Definition 1. *Let $R \subset V \times W$ be a binary relation and $L_R = \{v \in V \mid \exists w \in W : (v, w) \in R\}$ its language. Let Σ be a collection of PPT algorithms $\Sigma.\mathsf{ann}$, $\Sigma.\mathsf{res}$, $\Sigma.\mathsf{sim}$, $\Sigma.\mathsf{ext}$, and polynomial time predicate $\Sigma.\mathsf{ver}$. Let C be a finite set called the* challenge space*. Then Σ is a* Σ-protocol *for relation R if:*

[1] Here, we use the optimised multiplication protocol from [DN03].

Σ-Protocol 1 Σ_{PK}: Proof of plaintext knowledge

```
[Relation] $R = \{(X, (x, r)) \mid X = (1+N)^x r^N\}$
[Announcement] $\Sigma.\mathsf{ann}(X, (x, r)) :=$
    $a \in_R \mathbb{Z}_N; u \in_R \mathbb{Z}_N^*; B := (1+N)^a u^N;$ return $(B, (a, u))$
[Response] $\Sigma.\mathsf{res}(X, (x, r), B, (a, u), c) :=$
    $t := \lfloor (a + cx)/N \rfloor; d := a + cx; e := ur^c (1+N)^t;$ return $(d, e)$
[Simulator] $\Sigma.\mathsf{sim}(X, c) :=$
    $d \in_R \mathbb{Z}_N; e \in_R \mathbb{Z}_N^*; B := (1+N)^d e^N X^{-c};$ return $(B, c, (d, e))$
[Extractor] $\Sigma.\mathsf{ext}(X, B, c, c', (d, e), (d', e')) :=$
    $\beta, m := \langle$values such that $\beta(c - c') - mN = 1\rangle$
    return $((d - d')\beta, (e/e')^\beta X^{-m})$
[Verification] $\Sigma.\mathsf{ver}(X, B, c, (d, e)) := (1+N)^d e^N \overset{?}{=} BX^c$
```

Completeness *If* $(a, s) \leftarrow \Sigma.\mathsf{ann}(v, w)$*,* $c \in C$*, and* $r \leftarrow \Sigma.\mathsf{res}(v, w, a, s, c)$*, then* $\Sigma.\mathsf{ver}(v, a, c, r)$*.*

Special soundness *If* $v \in V$*,* $c \neq c'$*,* $\Sigma.\mathsf{ver}(v, a, c, r)$*, and* $\Sigma.\mathsf{ver}(v, a, c', r')$*, then* $w \leftarrow \Sigma.\mathsf{ext}(v, a, c, c', r, r')$ *satisfies* $(v, w) \in R$*.*

Special honest-verifier zero-knowledge *If* $v \in L_R$*,* $c \in C$*, then* $(a, r) \leftarrow \Sigma.\mathsf{sim}(v, c)$ *has the same probability distribution as* (a, r) *obtained by* $(a, s) \leftarrow \Sigma.\mathsf{ann}(v, w)$*,* $r \leftarrow \Sigma.\mathsf{res}(v, w, a, s, c)$*. If* $v \notin L_R$*, then* $(a, r) \leftarrow \Sigma.\mathsf{sim}(v, c)$ *satisfies* $\Sigma.\mathsf{ver}(v, a, c, r)$*.*

Non-Triviality *If* $(a, r) \leftarrow \Sigma.\mathsf{sim}(v, c)$*,* $(a', r') \leftarrow \Sigma.\mathsf{sim}(v, c)$ *then with overwhelming probability* $a \neq a'$*.*

Completeness states that a protocol between a honest prover and verifier succeeds; special soundness states that there exists an extractor $\Sigma.\mathsf{ext}$ that can extract a witness from two conversations with the same announcement; and special honest-verifier zero-knowledge states that there exists a simulator $\Sigma.\mathsf{sim}$ that can generate transcripts with the same distribution as full protocol runs without knowing the witness. Non-triviality essentially states that announcements are random from an exponential space; it is not a standard requirement of Σ-protocols, but most protocols satisfy it; and the Fiat-Shamir heuristic requires it [AABN08].

The CDN protocol uses a sub-protocol in which multiple parties simultaneously provide proofs based on the same challenge, called the "multiparty Σ-protocol". Namely, suppose each party from a set P wants to prove knowledge of a witness for a statement $v_i \in L_R$ with some Σ-protocol. To achieve this, each party in P broadcasts a commitment to its announcement; then, the computation parties jointly generate a challenge; and finally, all parties in P broadcast their response to this challenge, along with an opening of their commitment. The multiparty Σ-protocol is used as a building block in the CDN protocol by constructing a simulator that provides proofs on behalf of honest parties without knowing their witnesses ("zero knowledge"), and extracts witnesses from corrupted parties that give correct proofs ("soundness").

The CDN protocol uses three Σ-protocols: Σ_{PK} proving plaintext knowledge, Σ_{CM} proving correct multiplication, and Σ_{CD} proving correct decryption. The first two are due to [CDN01] (which also proves that they are Σ-protocols). Σ_{PK} (Σ-Protocol 1) proves knowledge of x, r such that $X = (1+N)^x r^N$ is an encryption of x with randomness r. Σ_{CM} (Σ-Protocol 2 in Appendix A) proves knowledge of (y, r, s) for (X, Y, Z) such that $Y = (1+N)^y r^N$ is an encryption of y with randomness r and $Z = X^y s^N$ is an encryption of the product of the plaintexts of X and Y randomised with s.

The proof of correct decryption is due to [Jur03]. In the threshold variant of Paillier encryption due to Damgård and Jurik [DJ01,Jur03], safe primes $p = 2p' + 1, q = 2q' + 1$ are used for the RSA modulus $N = pq$. Key generation involves generating a secret value d such that, given $c' = c^{4\Delta^2 d}$, anybody can compute the plaintext of c by "decoding" c' as $\mathsf{paillierdecode}(c') := ((c' - 1) \div N)(4\Delta^2)^{-1} \bmod N$. Here, $\Delta = n!$ and $\div$ denotes division as integers (using $N | c' - 1$). The value d is then (t, n) Shamir-shared modulo $Np'q'$ between the computation parties as shares s_i. Threshold decryption is done by letting t parties each compute $c_i = c^{2\Delta s_i}$; the value $c^{4\Delta^2 d}$ is obtained by applying Shamir reconstruction "in the exponent". Correct decryption is proved with respect to a public set of verification values. Namely, the public key includes values v, $v_0 = v^{\Delta^2 d}$, and $v_i = v^{\Delta s_i}$ for all computation parties $i \in \mathcal{P}$. Parties prove correctness of their decryption shares c_i of c by proving knowledge of $\Delta s_i = \log_{c^4}(c_i^2) = \log_v(v_i)$ for (c, c_i, v, v_i) using Σ-protocol Σ_{CD} (Σ-protocol 3 in Appendix A). (In the same way, the value v_0 can be used to prove correctness of c' with respect to c using a single instance of Σ_{CD}.)

2.3 Security of the CDN Protocol

In [CDN01], it is shown that the CDN protocol implements secure function evaluation in Canetti's non-concurrent model [Can98] if only a minority of computation parties are corrupted. Essentially, this means that in this case, the computation succeeds; the result is correct; and the honest parties' inputs remain private. This conclusion is true assuming honest set-up and security of the Paillier encryption scheme and the trapdoor commitment scheme used. If a majority of computation parties is corrupted, then because threshold $\lceil n/2 \rceil$ is used for the threshold cryptosystem, privacy is broken. As noted [ST06,IPS09], this can be remedied by raising the threshold, but in that case, the corrupted parties can make the computation break down at any point by refusing to cooperate. In Section 4.1, we present a variant of this model in which we prove the security of our protocols (using random oracles but no trapdoor commitments).

3 Multiparty Non-Interactive Proofs

In this section, we show how to produce non-interactive zero-knowledge proofs in a multiparty way. At several points in the above CDN protocol, all parties from a set P prove knowledge of witnesses for certain statements; the computation parties are convinced that those parties that succeed, do indeed know a witness. In CDN, these proofs are interactive; but for universal verifiability, we need non-interactive proofs that convince any third party. The traditional method to make

Protocol 1 FΣ: The Fiat-Shamir Heuristic

```
1. // pre: $\Sigma$ is a $\Sigma$-protocol, $P$ is a set of non-failed parties ($P \cap F = \emptyset$);
2. //          $v_P = \{v_i\}_{i \in P}$ are statements; $w_P = \{w_i\}_{i \in P}$ are witnesses
3. //          such that $(v_i, w_i)$ in the language of $\Sigma$ for all $i \in P$
4. // post: for all $i \in P \setminus F$, $\pi_i$ is Fiat-Shamir PoK of witness for $v_i$
5. // invariant: $F \subset C$: set of failed parties only includes corrupted parties
6. $\{\pi_i\}_{i \in P \setminus F} \leftarrow \mathrm{F}\Sigma(\Sigma, P, v_P, w_P) :=$
7.    foreach party $i \in P$ do
8.       $(a_i, s_i) := \Sigma.\mathsf{ann}(v_i, w_i); c_i := \mathcal{H}(v_i||a_i||i); r_i := \Sigma.\mathsf{res}(v_i, w_i, a_i, s_i, c_i)$
9.       $\mathsf{bcast}((a_i, c_i, r_i))$
10.   $F := F \cup \{i \in P \mid \neg \mathsf{fsver}_\Sigma(v_i, a_i, c_i, r_i, i)\}$
11.   return $\{(a_i, c_i, r_i)\}_{i \in P \setminus F}$
```

proofs non-interactive is the Fiat-Shamir heuristic; in Section 3.1, we outline it, and show that it is problematic in a multiparty setting. In Section 3.2, we present a new, "multiparty" Fiat-Shamir heuristic that works in our setting, and has the advantage of achieving smaller proofs by "homomorphically combining" the proofs of individual parties. In the remainder, $C \subset \mathcal{I} \cup \mathcal{P} \cup \{\mathcal{R}, \mathcal{V}\}$ denotes the set of corrupted parties; and F denotes the set of parties who failed to provide a correct proof when needed; this only happens for corrupted parties, so $F \subset C$.

Our results are in the random oracle model [BR93,Wee09], an idealised model of hash functions. In this model, evaluations of the hash function $\mathcal{H}$ are modelled as queries to a "random oracle" $\mathcal{O}$ that evaluates a perfectly random function. When simulating an adversary, a simulator can intercept these oracle queries and answer them at will, as long as the answers look random to the adversary. Security in the random oracle model does not generally imply security in the standard model [GK03], but it is often used because it typically gives simple, efficient protocols, and its use does not seem to lead to security problems in practice [Wee09]. See Appendix B for a detailed discription of our use of random oracles; and Section 5 for a discussion of the real-world implications of the particular flavour of random oracles we use.

3.1 The Fiat-Shamir Heuristic and Witness-Extended Emulation

The obvious way of making the proofs in the CDN protocol non-interactive, is to apply the Fiat-Shamir heuristic to all individual Σ-protocols. That is, party $i \in P$ proves knowledge of a witness for statement v by generating announcement a using $\Sigma.\mathsf{ver}$; setting challenge $c = \mathcal{H}(v||a||i)$[2]; and computing response r with $\Sigma.\mathsf{ver}$; a verifier accepts those proofs (a, c, r) for which $\mathsf{fsver}_\Sigma(v, a, c, r, i)$ holds, where $\mathsf{fsver}_\Sigma(v, a, c, r, aux)$ is defined as $\mathcal{H}(v||a||aux) = c \wedge \Sigma.\mathsf{ver}(v, a, c, r)$ (Protocol 1).

[2] Including the prover's identity prevents corrupted parties from replaying proofs by honest parties. Technically, if corrupted parties could replay honestly generated proofs, then the soundness property below would be broken because witnesses for these proofs cannot be extracted by rewinding the adversary to the point of the oracle query and reprogramming the random oracle.

Recall that security proofs require a simulator that simulates proofs of honest parties (zero-knowledge) and extracts witnesses of corrupted parties (soundness). In the random oracle model, Fiat-Shamir proofs for honest parties can be simulated by simulating a Σ-protocol transcript (a, c, r) and programming the random oracle so that $\mathcal{H}(v||a||i) = c$. Witnesses of corrupted parties can be extracted by rewinding the adversary to the point where it made an oracle query for $v||a||i$ and supplying a different value; but, as we show in Appendix B, this extraction can make the simulator very inefficient. In fact, if Fiat-Shamir proofs take place in R different rounds, then extracting witnesses may increase the running time of the simulator by a factor $O(R!)$. The reason is that the oracle query for a proof in one round may have in fact already been made in a previous round, in which case rewinding the adversary to extract one witness requires recursively extracting witnesses for all intermediate rounds. Hence, we can essentially only use the Fiat-Shamir heuristic in a constant number of rounds.

Moreover, in the CDN protocol, applying the Fiat-Shamir heuristic to each individual proof has the disadvantage that the verifier needs to check a number of proofs that depends linearly on the number of computation parties. In particular, for each multiplication gate, the verifier needs to check n proofs of correct multiplication and t proofs of correct decryption. Next, we show that we can avoid both the technical problems with witness extended emulation and the dependence on the number of computation parties by letting the computation parties collaboratively produce "combined proofs". (As discussed in Appendix B, there are other ways of just solving the technical problems with witness extended emulation, but they are not easier than the method we propose.)

3.2 Combined Proofs with the Multiparty Fiat-Shamir Heuristic

The crucial observation that allows parties to produce non-interactive zero-knowledge proofs collaboratively (e.g., [Des93,KMR12]) is that, for many Σ-protocols, transcripts of proofs with the same challenge can be "homomorphically combined". For instance, consider the classical Σ-protocol for proving knowledge of a discrete logarithm due to Schnorr [Sch89]. Suppose we have two Schnorr transcripts proving knowledge of $x_1 = \log_g h_1$, $x_2 = \log_g h_2$, i.e., two tuples (a_1, c, r_1) and (a_2, c, r_2) such that $g^{r_1} = a_1(h_1)^c$ and $g^{r_2} = a_2(h_2)^c$. Then $g^{r_1+r_2} = (a_1a_2)(h_1h_2)^c$, so $(a_1a_2, e, r_1 + r_2)$ is a Schnorr transcript proving knowledge of discrete logarithm $x_1 + x_2 = \log_g(h_1h_2)$. For our purposes, we demand that such homomorphisms satisfy two properties. First, when transcripts of at least $\lceil n/2 \rceil$ parties are combined, the result is a valid transcript (the requirement of having at least $\lceil n/2 \rceil$ transcripts is needed for decryption proofs to ensure that there are enough decryption shares). Second, when fewer than $\lceil n/2 \rceil$ parties are corrupted, the combination of different honest announcements with the same corrupted announcements is likely to lead to a different combined announcement. This helps to eliminate the rewinding problems for Fiat-Shamir discussed above.

Definition 2. *Let Σ be a Σ-protocol for relation $R \subset V \times W$. Let Φ be a collection of partial functions $\Phi.\mathsf{stmt}$, $\Phi.\mathsf{ann}$, and $\Phi.\mathsf{resp}$. We call Φ a* homomorphism *of Σ if:*

Protocol 2 MΣ: The Multi-Party Fiat-Shamir Heuristic

```
// pre: $\Sigma$ is a $\Sigma$-protocol with homomorphism $\Phi$, $P$ is a set of non-failed
//      parties ($P \cap F = \emptyset$), $v_P = \{v_i\}_{i \in P}$ statements w/ witnesses $w_P = \{w_i\}_{i \in P}$
// post: if $|P \setminus F| \geq \lceil n/2 \rceil$, then $v_{P \setminus F}$ is the combined statement
//       $\Phi.\mathsf{stmt}(\{v_i\}_{i \in P \setminus F})$, and $\pi_{P \setminus F}$ is a corresponding Fiat-Shamir proof
// invariant: $F \subset C$: set of failed parties only includes corrupted parties
$(v_{P \setminus F}, \pi_{P \setminus F}) \leftarrow \mathrm{M}\Sigma(\Sigma, \Phi, P, v_P, w_P, aux) :=$
  repeat
    foreach party $i \in P \setminus F$ do
      $(a_i, s_i) := \Sigma.\mathsf{ann}(v_i, w_i); h_i := \mathcal{H}(a_i || i); \mathsf{bcast}(h_i)$
    foreach party $i \in P \setminus F$ do $\mathsf{bcast}(a_i)$
    $F' := F; F := F \cup \{i \in P \setminus F \mid h_i \neq \mathcal{H}(a_i || i)\}$
    if $F = F'$ then                    // all parties left provided correct hashes
      $c := \mathcal{H}(\Phi.\mathsf{stmt}(\{v_i\}_{i \in P \setminus F}) || \Phi.\mathsf{ann}(\{a_i\}_{i \in P \setminus F}) || aux)$
      foreach party $i \in P \setminus F$ do $r_i := \Sigma.\mathsf{res}(v_i, w_i, a_i, s_i, c); \mathsf{bcast}(r_i)$
      $F := F \cup \{i \in P \setminus F \mid \neg \Sigma.\mathsf{ver}(v_i, a_i, c, r_i)\}$
      if $F = F'$ then                  // all parties left provided correct responses
        return $(\Phi.\mathsf{stmt}(\{v_i\}_{i \in P \setminus F}),$
                   $(\Phi.\mathsf{ann}(\{a_i\}_{i \in P \setminus F}), c, \Phi.\mathsf{resp}(\{r_i\}_{i \in P \setminus F})))$
  until $|P \setminus F| < \lceil n/2 \rceil$              // until not enough parties left
  return $(\perp, \perp)$
```

Combination *Let c be a challenge; I a set of parties such that $|I| \geq \lceil n/2 \rceil$; and $\{(v_i, a_i, r_i)\}_{i \in I}$ a collection of statements, announcements, and responses. If $\Phi.\mathsf{stmt}(\{v_i\}_{i \in I})$ is defined and for all i, $\Sigma.\mathsf{ver}(v_i, a_i, c, r_i)$ holds, then also $\Sigma.\mathsf{ver}(\Phi.\mathsf{stmt}(\{v_i\}_{i \in I}), \Phi.\mathsf{ann}(\{a_i\}_{i \in I}), c, \Phi.\mathsf{resp}(\{r_i\}_{i \in I}))$.*

Randomness *Let c be a challenge; $C \subset I$ sets of parties such that $|C| < \lceil n/2 \rceil \leq |I|$; $\{v_i\}_{i \in I}$ statements s.t. $\Phi.\mathsf{stmt}(\{v_i\}_{i \in I})$ is defined; and $\{a_i\}_{i \in I \cap C}$ announcements. If $(a_i, \_), (a'_i, \_) \leftarrow \Sigma.\mathsf{sim}(v_i, c)$ $\forall i \in I \setminus C$, then with overwhelming probability, $\Phi.\mathsf{ann}(\{a_i\}_{i \in I}) \neq \Phi.\mathsf{ann}(\{a_i\}_{i \in I \cap C} \cup \{a'_i\}_{i \in I \setminus C})$.*

Given a Σ-protocol with homomorphism Φ, parties holding witnesses $\{w_i\}$ for statements $\{v_i\}$ can together generate a Fiat-Shamir proof $(a, \mathcal{H}(v||a||aux), r)$ of knowledge of a witness for the "combined statement" $v = \Phi.\mathsf{stmt}(\{v_i\})$. Namely, the parties each provide announcement a_i for their own witness; compute $a = \Phi.\mathsf{ann}(\{a_i\})$ and $\mathcal{H}(v||a||aux)$; and provide responses r_i. Taking $r = \Phi.\mathsf{resp}(\{r_i\})$, the combination property from the above definition guarantees that we indeed get a validating proof. However, we cannot simply let the parties broadcast their announcements in turn, because to prove security in that case, the simulator needs to provide the announcements for the honest parties without knowing the announcements of the corrupted parties, hence without being able to program the random oracle on the combined announcement. We solve this by starting with a round in which each party commits to its announcement (the same trick was used in a different setting in [NKDM03])[3].

[3] As in [NKDM03], it may be possible to remove the additional round under the non-standard known-target discrete log problem.

The *multiparty Fiat-Shamir heuristic* (Protocol 2) let parties collaboratively produce Fiat-Shamir proofs based on the above ideas. Apart from the above procedure (lines 8, 9, 10, 13, and 14), the protocol also contains error handling. Namely, we throw out parties that provide incorrect hashes to their announcements (line 11) or incorrect responses (line 15). If we have correct responses for all correctly hashed announcements, then we apply the homomorphism (line 17–18); otherwise, we try again with the remaining parties. If the number of parties drops below $\lceil n/2 \rceil$, the homomorphism can no longer be applied, so we return with an error (line 20). Note that, as in the normal Fiat-Shamir heuristic, the announcements do not need to be stored if they can be computed from the challenge and response (as will be the case for the Σ-protocols we consider).

In the CDN protocol, the multiparty Fiat-Shamir heuristic allows us to obtain a proof that multiplication was done correctly that is independent of the number of computation parties. Recall that, for multiplication of encryptions X of x and Y of y, each computation party provides encryptions D_i of d_i and E_i of $d_i \cdot y$, and proves that E_i encrypts the product of the plaintexts of Y and D_i; and each computation party provides decryption share S_i of encryption $XD_1 \cdots D_n$, and proves it correct. As we show in Section A, the multiplication proofs can be combined with homomorphism Φ_{CM} into one proof that $\prod E_i$ encrypts the product of the plaintexts of Y and $\prod D_i$; and the decryption proofs can be combined with homomorphism Φ_{CD} into one proof that a combination S_0 of the decryption shares is correct. In the CDN protocol, the individual D_i, E_i, and S_i are not relevant, so also the combined values convince a verifier of correct multiplication.

Concerning security, recall that we need a simulator that simulates proofs of honest parties without their witnesses (zero-knowledge) and extracts the witnesses of corrupted parties (soundness). In Appendix C, we present such a simulator. Essentially, it "guesses" the announcements of the corrupted parties based on the provided hashes; then simulates the Σ-protocol for the honest parties; and programs the random oracle on the combined announcement. It obtains witnesses for the corrupted parties by rewinding to just before the honest parties provide their announcements: this way, the corrupted parties are forced to use the announcements that they provided the hashes of (hence special soundness can be invoked), whereas the honest parties can provide new simulated announcements by reprogramming the random oracle. The simulator requires that fewer than $\lceil n/2 \rceil$ provers are corrupted so that we can use the randomness property of the Σ-protocol homomorphism (Definition 2). (When more than $\lceil n/2 \rceil$ provers are corrupted, we use an alternative proof strategy that uses witness-extended emulation instead of this simulator.)

4 Universally Verifiable MPC

In the previous section, we have shown how to produce non-interactive zero-knowledge proofs in a multiparty way. We now use this observation to obtain universally verifiable MPC. We first define security for universally verifiable MPC; and then obtain universally verifiable MPC by adapting the CDN protocol.

ITM 1 $\mathcal{T}_{\text{VSFE}}$: trusted party for verifiable secure function evaluation

```
1. // compute $f$ on $\{x_i\}_{i\in\mathcal{I}}$ for $\mathcal{R}$ with corrupted parties $C$; $\mathcal{V}$ learns encryption
2. $\mathcal{T}_{\text{VSFE}}(C,(N,v,v_0,\{v_i\}_{i\in\mathcal{P}})) :=$
3.     // input phase
4.     forall $i \in \mathcal{I} \setminus C$ do  $x_i := \mathsf{recv}(\mathcal{I}_i)$                // honest inputs
5.     if $|\mathcal{P}\cap C| \geq \lceil n/2 \rceil$ then $\mathsf{send}(\{x_i\}_{i\in\mathcal{I}\setminus C},\mathcal{S})$   // send to corrupted majority
6.     $\{x_i\}_{i\in\mathcal{I}\cap C} := \mathsf{recv}(\mathcal{S})$                        // corrupted inputs
7.     // computation phase
8.     $r := f(x_1,\ldots,x_m)$
9.     // output phase
10.    if $\mathcal{R} \notin C$ then      // honest $\mathcal{R}$: adversary learns encryption, may block result
11.        $s \in_R \mathbb{Z}_N^*$; $R := (1+N)^r s^N$; $\mathsf{send}(R,\mathcal{S})$
12.        if $|\mathcal{P}\cap C| \geq \lceil n/2 \rceil$ and $\mathsf{recv}(\mathcal{S}) = \perp$ then $r := \perp$; $s := \perp$; $R := \perp$
13.        $\mathsf{send}((r,s),\mathcal{R})$
14.    else            // corrupted $\mathcal{R}$: adversary learns output, may block result to $\mathcal{V}$
15.        $\mathsf{send}(r,\mathcal{S})$; $s := \mathsf{recv}(\mathcal{S})$
16.        if $s = \perp$ then $R := \perp$ else $R := (1+N)^r s^N$
17.    // proof phase
18.    if $\mathcal{V} \notin C$ then $\mathsf{send}(R,\mathcal{V})$
```

4.1 Security Model for Verifiable MPC

Our security model is an adaptation of the model of [Can98,CDN01] to the setting of universal verifiability in the random oracle model. We first explain the general execution model, which is as in [Can98,CDN01] but with a random oracle added; we then explain how to model verifiability in this execution model as the behaviour of the ideal-world trusted party. The general execution model compares protocol executions in the real and ideal world.

In the real world, a protocol π between m input parties $i \in \mathcal{I}$, n computation parties $i \in \mathcal{P}$, a result party $\mathcal{R}$ and a verifier $\mathcal{V}$ is executed on an open broadcast network with rushing in the presence of an active static adversary $\mathcal{A}$ corrupting parties $C \subset \mathcal{I} \cup \mathcal{P} \cup \{\mathcal{R},\mathcal{V}\}$. The protocol execution starts by incorruptibly setting up the Paillier threshold cryptosystem, i.e., generating public key $\mathsf{pk} = (N,v,v_0,\{v_i\}_{i\in\mathcal{P}})$ with RSA modulus N and verification values v, v_0, v_i, and secret key shares $\{s_i\}_{i\in\mathcal{P}}$ (see Section 2.2). Each input party $i \in \mathcal{I}$ gets input (pk, x_i); each computation party $i \in \mathcal{P}$ gets input (pk, s_i); and the result party $\mathcal{R}$ gets input pk. The adversary gets the inputs $(\mathsf{pk},\{x_i\}_{i\in\mathcal{I}\cap C},\{s_i\}_{i\in\mathcal{P}\cap C})$ of the corrupted parties, and has an auxiliary input a. During the protocol, parties can query the random oracle; the oracle answers new queries randomly, and repeated queries consistently. At the end of the protocol, each honest party outputs a value according to the protocol; the corrupted parties output $\perp$; and the adversary outputs a value at will. Define $\text{EXEC}_{\pi,\mathcal{A}}(k,(x_1,\ldots,x_m),C,a)$ to be the random variable, given security parameter k, consisting of the outputs of all parties (including the adversary) and the set $\mathcal{O}$ of oracle queries and responses.

The ideal-world execution similarly involves m input parties $i \in \mathcal{I}$, n computation parties $i \in \mathcal{P}$, result party $\mathcal{R}$, verifier $\mathcal{V}$, and an adversary $\mathcal{S}$ corrupting parties $C \subset \mathcal{I} \cup \mathcal{P} \cup \{\mathcal{R}, \mathcal{V}\}$; but now, there is also an incorruptible trusted party $\mathcal{T}$. As before, the execution starts by setting up the keys $(\mathsf{pk}, \{s_i\}_{i \in \mathcal{P}})$ of the Paillier cryptosystem. The input parties receive x_i as input; the trusted party receives a list C of corrupted parties and the public key pk. Then, it runs the code $\mathcal{T}_{\mathrm{VSFE}}$ shown in ITM 1, which we explain later. The adversary gets inputs $(\mathsf{pk}, C, \{x_i\}_{i \in \mathcal{I} \cap C}, \{s_i\}_{i \in \mathcal{P} \cap C})$, and outputs a value at will. In this model, there is no random oracle; instead, the adversary chooses the set $\mathcal{O}$ of oracle queries and responses (typically, those used to simulate a real-world adversary). As in the real-world case, define $\mathrm{IDEAL}_{\mathcal{T}_{\mathrm{SFE}}, \mathcal{S}}(k, (x_1, \ldots, x_m), C, a)$ as the random variable, given security parameter k, consisting of all parties' outputs and $\mathcal{O}$.

Definition 3. *Protocol* π implements verifiable secure function evaluation in the random oracle model *if, for every real-world adversary* $\mathcal{A}$*, there exists an ideal-world adversary* $\mathcal{S}_{\mathcal{A}}$ *such that, for all inputs* $x_1, \ldots, x_m$*; all sets of corrupted parties* C*; and all auxiliary input* a*:* $EXEC_{\pi, \mathcal{A}}(k, (x_1, \ldots, x_m), C, a)$ *and* $IDEAL_{\mathcal{T}_{VSFE}, \mathcal{S}_{\mathcal{A}}}(k, (x_1, \ldots, x_m), C, a)$ *are computationally indistinguishable in security parameter* k*.*

We remark that, while security in non-random-oracle secure function evaluation [Can98,CDN01] is preserved under (subroutine) composition, this is not the case for our random oracle variant. The reason is that our model and protocols assume that the random oracle is not used outside of the protocol. Using the random oracle model with dependent auxiliary input [Unr07,Wee09] might be enough to obtain a composition property; but adaptations are needed to make our protocol provably secure in that model. See Section 5 for a discussion.

We now discuss the trusted party $\mathcal{T}_{\mathrm{VSFE}}$ for verifiable secure function evaluation. Whenever the computation succeeds, $\mathcal{T}_{\mathrm{VSFE}}$ guarantees that the results are correct. Namely, $\mathcal{T}_{\mathrm{VSFE}}$ sends the result r of the computation and randomness s to $\mathcal{R}$ (line 11), and it sends encryption $(1+N)^r s^N$ of the result with randomness s to $\mathcal{V}$ (line 18); if the computation failed, $\mathcal{R}$ gets $(\perp, \perp)$ and $\mathcal{V}$ gets $\perp$.[4] Whether $\mathcal{T}_{\mathrm{VSFE}}$ guarantees privacy (i.e., only $\mathcal{R}$ can learn the result) and robustness (i.e., the computation does not fail) depends on which parties are corrupted. Privacy and robustness with respect to $\mathcal{R}$ are guaranteed as long as only a minority of computation parties are corrupted. If not, then in line 5, $\mathcal{T}_{\mathrm{VSFE}}$ sends the honest parties' inputs to the adversary; and in line 12, it gives the adversary the option to block the computation by sending $\perp$. For robustness with respect to $\mathcal{V}$, moreover, the result party needs to be honest. If not, then in line 16, $\mathcal{T}_{\mathrm{VSFE}}$ gives the adversary the option to block $\mathcal{V}$'s result by sending $\perp$; in

[4] Although we only guarantee computational indistinguishability and the verifier does not know what value is encrypted, this definition *does* guarantee that $\mathcal{V}$ receives the correct result. This is because the ideal-world output of the protocol execution contains $\mathcal{R}$'s r and s and $\mathcal{V}$'s $(1+N)^r s^N$, so a distinguisher between the ideal and real world can check correctness of $\mathcal{V}$'s result. (If s were not in $\mathcal{R}$'s result, this would not be the case, and correctness of $\mathcal{V}$'s result would *not* be guaranteed.) Also, note that although privacy depends on the security of the encrytion scheme, correctness *does not rely on any knowledge assumption.*

any case, it can choose the randomness. (Note that these thresholds are specific to CDN's "honest majority" setting; e.g., other protocols may satisfy privacy if all computation parties except one are corrupted.)

Note that this model does not cover the "universality" aspect of universally verifiable MPC. This is because the security model for secure function evaluation only covers the input/output behaviour of protocols, not the fact that "the verifier can be anybody". Hence, we design universally verifiable protocols by proving that they are verifiable, and then arguing based on the characteristics of the protocol (e.g., the verifier does not have any secret values) that this verifiability is "universal".

4.2 Universally Verifiable CDN

We now present the UVCDN protocol (Protocol 3) for universally verifiable secure function evaluation.

First, a full run of the CDN protocol (Section 2.1) is performed, with all proofs made non-interactive (lines 5–36). As discussed, we can use the normal Fiat-Shamir (FS) heuristic in only a constant number of rounds; and we can use the multiparty FS heuristic only when it gives a "combined statement" that makes sense. This leads to the following choices. During the *input phase* of the protocol, the input parties provide their inputs with a proof of knowledge using the normal FS heuristic (lines 6–8). During the *computation phase*, the function is evaluated gate-by-gate; for multiplication gates, the multiplication protocol from [DN03] is used, with proofs of correct multiplication and decryption using the multiparty FS heuristic (lines 15–26). During the *output phase*, the result party obtains the result by broadcasting an encryption of a random d and proving knowledge using the normal FS heuristic (lines 29–30); the computation parties decrypt the result plus d, proving correctness using the multiparty FS heuristic (lines 34–35). From this, the result party learns result r (line 36); and it knows the intermediate values from the protocol and the proofs showing they are correct.

Finally, during the *proof phase*, the result party sends these intermediate values and proofs to the verifier (line 40). The verifier runs procedure vercomp (Algorithm 1) to verify the correctness of the computation. The inputs to this verification procedure are the public key of the Paillier cryptosystem; the encrypted inputs $\{X_i\}_{i \in \mathcal{I}}$ by the input parties; and the proof π by the result party (which consists of proofs for each multiplication gate, and the two proofs from the output phase of the protocol). The verifier checks the proofs for each multiplication gate from the computation phase (lines 6–14); and the proofs from the output phase (lines 16–20), finally obtaining an encryption of the result (line 21). While not specified in vercomp, the verifier does also verify the proofs from the input phase: namely, in lines 6–8 of UVCDN, the verifier receives encrypted inputs and verifies their proofs to determine the encrypted inputs $\{X_i\}_{i \in \mathcal{I}}$ of the computation.

Apart from checking the inputs during the input phase, the verifier does not need to be present for the remainder of the computation until receiving π from $\mathcal{R}$. This is what makes verification "universal": in practice, we envision that a trusted party publicly announces the Paillier public keys, and the input parties

Protocol 3 UVCDN: universally verifiable CDN

```
// **pre**: $\mathsf{pk}/\{s_i\}_{i\in\mathcal{P}}$ threshold Paillier public/secret keys, $\{x_i\}_{i\in\mathcal{I}}$ function input
// **post**: output $R$ according to ideal functionality ITM 1
$R \leftarrow \mathrm{UVCDN}(\mathsf{pk} = (N, v, v_0, \{v_i\}_{i\in\mathcal{P}}), \{s_i\}_{i\in\mathcal{P}}, \{x_i\}_{i\in\mathcal{I}}) :=$
    // input phase
    $F := \emptyset$
    **foreach party** $i \in \mathcal{I}$ **do** $r_i \in_R \mathbb{Z}_N^*; X_i = (1+N)^{x_i} r_i^N; \mathsf{bcast}(X_i)$
    $\{\pi_{\mathrm{PK}i}\}_{i\in\mathcal{I}\setminus F} := \mathrm{F}\Sigma(\Sigma_{\mathrm{PK}}, \mathcal{I}, \{X_i\}_{i\in\mathcal{I}}, \{(x_i, r_i)\}_{i\in\mathcal{I}})$
    **foreach** $i \in \mathcal{I} \cap F$ **do** $X_i := 1$
    // computation phase
    **foreach** gate **do**
        **if** $\langle$constant gate $c$ with value $v\rangle$ **then** $X_c := (1+N)^v$
        **if** $\langle$addition gate $c$ with inputs $a$, $b\rangle$ **then** $X_c := X_a X_b$
        **if** $\langle$subtraction gate $c$ with inputs $a$, $b\rangle$ **then** $X_c := X_a X_b^{-1}$
        **if** $\langle$multiplication gate $c$ with inputs $a$, $b\rangle$ **then**    // [DN03] multiplication
            **foreach party** $i \in \mathcal{P} \setminus F$ **do**
                $d_i \in_R \mathbb{Z}_N; r_i, t_i \in_R \mathbb{Z}_N^*; D_i := (1+N)^{d_i} r_i^N; E_i := (X_b)^{d_i} t_i^N$
                $\mathsf{bcast}(D_i, E_i)$
            $((\_, D_c, E_c), \pi_{\mathrm{CM}c}) :=$
                    $\mathrm{M}\Sigma(\Sigma_{\mathrm{CM}}, \Phi_{\mathrm{CM}}, \mathcal{P}\setminus F, \{(X_b, D_i, E_i)\}_{i\in\mathcal{P}\setminus F}, \{(d_i, r_i, t_i)\}_{i\in\mathcal{P}\setminus F})$
            **if** $|\mathcal{P}\setminus F| < \lceil n/2 \rceil$ **then break**
            $S_c := X_a \cdot D_c$
            **foreach party** $i \in \mathcal{P} \setminus F$ **do** $S_i := (S_c)^{2\Delta s_i}; \mathsf{bcast}(S_i)$
            $((\_, S_{0,c}, \_, \_), \pi_{\mathrm{CD}c}) :=$
                    $\mathrm{M}\Sigma(\Sigma_{\mathrm{CD}}, \Phi_{\mathrm{CD}}, \mathcal{P}\setminus F, \{(S_c, S_i, v, v_i)\}_{i\in\mathcal{P}\setminus F}, \{\Delta s_i\}_{i\in\mathcal{P}\setminus F})$
            **if** $|\mathcal{P}\setminus F| < \lceil n/2 \rceil$ **then break**
            $s := \mathsf{paillierdecode}(S_{0,c}); X_c := (X_b)^s \cdot E_c^{-1}$
    // output phase
    **if** $|\mathcal{P}\setminus F| \geq \lceil n/2 \rceil$ **then**
        **party** $\mathcal{R}$ **do** $d \in_R \mathbb{Z}_N; s \in_R \mathbb{Z}_N^*; D := (1+N)^d s^N; \mathsf{bcast}(D)$
        $\pi_{\mathrm{PK}d} := \mathrm{F}\Sigma(\Sigma_{\mathrm{PK}}, \{\mathcal{R}\}, \{D\}, \{(d, s)\})$
        **if** $\mathcal{R} \notin F$ **then**
            $Y := X_{\mathrm{outgate}} \cdot D^{-1}$
            **foreach party** $i \in \mathcal{P} \setminus F$ **do** $Y_i := Y^{2\Delta s_i}; \mathsf{bcast}(Y_i)$
            $((\_, Y_0, \_, \_), \pi_{\mathrm{CD}}, y) :=$
                    $\mathrm{M}\Sigma(\Sigma_{\mathrm{CD}}, \Phi_{\mathrm{CD}}, \mathcal{P}\setminus F, \{(Y, Y_i, v, v_i)\}_{i\in\mathcal{P}\setminus F}, \{\Delta s_i\}_{i\in\mathcal{P}\setminus F}, D)$
            **party** $\mathcal{R}$ **do** $y := \mathsf{paillierdecode}(Y_0); r := y + d$
    // proof phase
    **party** $\mathcal{R}$ **do**
        **if** $|\mathcal{P}\setminus F| \geq \lceil n/2 \rceil$                    // computation, delivery succeeded
            $\mathsf{send}((\{(D_c, E_c, \Pi_{\mathrm{CM}c}, S_{0,c}, \Pi_{\mathrm{CD}c})\}_{c\in\mathrm{gates}}, (D, \pi_{\mathrm{PK}d}, Y_0, \pi_{\mathrm{CD}y})), \mathcal{V})$
            **return** $(r, s)$
        **else**                                                  // computation failed
            $\mathsf{send}(\perp, \mathcal{V});$ **return**$(\perp, \perp)$
    **party** $\mathcal{V}$ **do** $\pi := \mathsf{recv}(\mathcal{R});$ **return** $\mathsf{vercomp}(\mathsf{pk}, \{X_i\}_{i\in\mathcal{I}}, \pi)$
    **foreach party** $i \in \mathcal{I} \cup \mathcal{P}$ **do return** $\perp$
```

Algorithm 1 vercomp: verifier's gate-by-gate verification of the computation

```
 1. // pre: pk public key, $\{X_i\}_{i\in\mathcal{I}}$ encryptions, $(\{\Pi_{\mathrm{mul}i}\}, \Pi_{\mathrm{result}})$ tuple
 2. // post: if $(\{\Pi_{\mathrm{mul}i}\}, \Pi_{\mathrm{result}})$ proves correctness of $Y$, $X_o = Y$; otherwise, $X_o = \perp$
 3. $X_o \leftarrow \mathsf{vercomp}(\mathsf{pk} = (N, v, v_0, \{v_i\}_{i\in\mathcal{P}}), \{X_i\}_{i\in\mathcal{I}}, (\{\Pi_{\mathrm{mul}i}\}, \Pi_{\mathrm{result}})) :=$
 4.     // verification of input phase: see lines 6–8 of UVCDN
 5.     // verification of computation phase
 6.     foreach gate do
 7.         if ⟨constant gate $c$ with value $v$⟩ then $X_c := (1+N)^v$
 8.         if ⟨addition gate $c$ with inputs $a$, $b$⟩ then $X_c := X_a X_b$
 9.         if ⟨subtraction gate $c$ with inputs $a$, $b$⟩ then $X_c := X_a X_b^{-1}$
10.         if ⟨multiplication gate $c$ with inputs $a$, $b$⟩ then
11.             $(D, E, (a, c, r), S_0, (a', c', r')) := \Pi_{\mathrm{mul}c};\ S := X_a \cdot D^{-1}$
12.             if $\neg\mathsf{fsver}_{\Sigma_{\mathrm{CM}}}((X_b, D, E), a, c, r)$ then return $\perp$
13.             if $\neg\mathsf{fsver}_{\Sigma_{\mathrm{CD}}}((S, S_0, v, v_0), a', c', r')$ then return $\perp$
14.             $s := \mathsf{paillierdecode}(S_0); X_c := (X_b)^s E^{-1}$
15.     // verification of output phase
16.     $(D, (a_{\mathrm{out}}, c_{\mathrm{out}}, r_{\mathrm{out}}), Y_0, (a_{\mathrm{dec}}, c_{\mathrm{dec}}, r_{\mathrm{dec}})) := \Pi_{\mathrm{result}}$
17.     if $\neg\mathsf{fsver}_{\Sigma_{\mathrm{PK}}}(D, a_{\mathrm{out}}, c_{\mathrm{out}}, r_{\mathrm{out}}, \mathcal{R})$ then return $\perp$
18.     $Y := X_{\mathrm{outgate}} \cdot D^{-1}$
19.     if $\neg\mathsf{fsver}_{\Sigma_{\mathrm{CD}}}((Y, Y_0, v, v_0), a_{\mathrm{dec}}, c_{\mathrm{dec}}, r_{\mathrm{dec}}, D)$ then return $\perp$
20.     $y := \mathsf{paillierdecode}(Y_0)$
21.     return $(1+N)^y D$                                   // encryption of $y + d = r$
```

publicly announce their encrypted inputs with associated proofs: then, anybody can use the verification procedure to verify if a given proof π is correct with respect to these inputs. In Appendix D, we prove that:

Theorem 1. *Protocol UVCDN implements verifiable secure function evaluation in the random oracle model.*

The proof uses two simulators: one for a honest majority of computation parties; one for a corrupted majority. The former simulator extends the one from [CDN01], obtaining privacy with a reduction to semantic security of the threshold Paillier cryptosystem. The latter does not guarantee privacy, and so can simulate the adversary by running the real protocol, ensuring correctness by witness-extended emulation.

5 Concluding Remarks

Our security model is specific to the CDN setting in two respects. First, we explicitly model that the verifier receives a Paillier encryption of the result (as opposed to another kind of encryption or commitment). We chose this formulation for concreteness; but our model generalises easily to other representations of the result. Second, it is specific to the setting where a minority of parties may be actively corrupted; but it is possible to change the model to other corruption models. For instance, it is possible to model the setting from [BDO14] where privacy is guaranteed when there is at least one honest computation party (and our protocols can be adapted to that setting). The combination of passively secure multiparty computation with universal verifiability is another interesting possible adaptation.

Our protocols are secure in the random oracle model "without dependent auxiliary input" [Wee09]. This means our security proofs assume that the random oracle has not been used before the protocol starts. Moreover, our simulator can only simulate logarithmically many sequential runs of our protocol due to technical limits of witness-extended emulation. These technical issues reflect the real-life problem that a verifier cannot see if a set of computation parties have just performed a computation, or they have simply replayed an earlier computation transcript. As discussed in [Unr07], both problems can be solved in practice by instantiating the random oracle with a keyed hash function, with every computation using a fresh random key. Note that all existing constructions require the random oracle model; achieving universally verifiable (or publicly auditable) multiparty computation in the standard model is open.

Several interesting variants of our protocol are possible. First, it is easy to achieve publicly auditable multiparty computation [BDO14] by performing a public decryption of the result rather than a private decryption for the result party. Another variant is basic outsourcing of computation, in which the result party does not need to be present at the time of the computation, but afterwards gets a transcript from which it can derive the computation result. Finally, it is possible to achieve universal verifiability using other threshold cryptosystems than Paillier. In particular, while the threshold ElGamal cryptosystem is much more efficient than threshold Paillier, it cannot be used directly with our protocols because it does not have a general decryption operation; but universally verifiable multiparty using ElGamal should still be possible by instead adapting the "conditional gate" variant of the CDN protocol from [ST04].

References

AABN08. M. Abdalla, J. H. An, M. Bellare, and C. Namprempre. From Identification to Signatures Via the Fiat-Shamir Transform: Necessary and Sufficient Conditions for Security and Forward-Security. *IEEE Transactions on Information Theory*, 54(8):3631–3646, 2008.

ACG$^+$14. P. Ananth, N. Chandran, V. Goyal, B. Kanukurthi, and R. Ostrovsky. Achieving Privacy in Verifiable Computation with Multiple Servers - Without FHE and without Pre-processing. In *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*, volume 8383 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 2014.

BCD$^+$09. P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. P. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure Multiparty Computation Goes Live. In *Proceedings of FC '09*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

BDO14. C. Baum, I. Damgård, and C. Orlandi. Publicly Auditable Secure Multi-Party Computation. In *Proceedings of SCN '14*, volume 8642 of *Lecture Notes in Computer Science*, pages 175–196. Springer, 2014.

BR93. M. Bellare and P. Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of CCS '93*, pages 62–73. ACM, 1993.

Can98. R. Canetti. Security and Composition of Multi-party Cryptographic Protocols. *Journal of Cryptology*, 13:2000, 1998.

CDN01. R. Cramer, I. Damgård, and J. Nielsen. Multiparty Computation from Threshold Homomorphic Encryption. In *Proceedings of EUROCRYPT '01*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–300. Springer, 2001.

CF85. J. Cohen and M. Fischer. A Robust and Verifiable Cryptographically Secure Election Scheme. In *Proceedings of FOCS '85*, pages 372–382. IEEE, 1985.

Des93. Y. Desmedt. Threshold cryptosystems. In *Proceedings of AUSCRYPT '92*, volume 718 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1993.

dH12. S. de Hoogh. *Design of large scale applications of secure multiparty computation: secure linear programming.* PhD thesis, Eindhoven University of Technology, 2012.

dHSCodA14. S. de Hoogh, B. Schoenmakers, P. Chen, and H. op den Akker. Practical Secure Decision Tree Learning in a Teletreatment Application. In *Proceedings of FC '14*, volume 8437 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2014.

DJ01. I. Damgård and M. Jurik. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In *Proceedings of PKC '01*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

DN03. I. Damgård and J. B. Nielsen. Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption. In *Proceedings of CRYPTO '03*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

DPSZ12. I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *Proceedings of CRYPTO '12*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

EFLL12. Y. Ejgenberg, M. Farbstein, M. Levy, and Y. Lindell. SCAPI: The Secure Computation Application Programming Interface. *IACR Cryptology ePrint Archive*, 2012:629, 2012.

FGP14. D. Fiore, R. Gennaro, and V. Pastro. Efficiently Verifiable Computation on Encrypted Data. In *Proceedings of CCS '14*, pages 844–855. ACM, 2014.

GK03. S. Goldwasser and Y. T. Kalai. On the (In)security of the Fiat-Shamir Paradigm. In *Proceedings of FOCS '03*, pages 102–113. IEEE Computer Society, 2003.

GKP$^+$13. S. Goldwasser, Y. T. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of STOC '13*, pages 555–564. ACM, 2013.

Gro04. J. Groth. Evaluating Security of Voting Schemes in the Universal Composability Framework. In *Proceedings of ACNS '04*, volume 3089 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2004.

IPS09. Y. Ishai, M. Prabhakaran, and A. Sahai. Secure Arithmetic Computation with No Honest Majority. In *Proceedings of TCC '09*, volume 5444 of *Lecture Notes in Computer Science*, pages 294–314. Springer, 2009.

Jur03. M. J. Jurik. *Extensions to the Paillier Cryptosystem with Applications to Cryptological Protocols*. PhD thesis, University of Aarhus, 2003.

KMR12. M. Keller, G. L. Mikkelsen, and A. Rupp. Efficient Threshold Zero-Knowledge with Applications to User-Centric Protocols. In *Proceedings of ICITS 2012*, volume 7412 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 2012.

NKDM03. A. Nicolosi, M. N. Krohn, Y. Dodis, and D. Mazières. Proactive Two-Party Signatures for User Authentication. In *Proceedings of NDSS 2003*. The Internet Society, 2003.

Pai99. P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Proceedings of EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

PHGR13. B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *Proceedings of S&P 2013*, pages 238–252. IEEE, 2013.

Sch89. C. Schnorr. Efficient Identification and Signatures for Smart Cards. In *Proceedings of CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252. Springer, 1989.

SK95. K. Sako and J. Kilian. Receipt-Free Mix-Type Voting Scheme—A Practical Solution to the Implementation of a Voting Booth. In *Proceedings of EUROCRYPT '95*, volume 921 of *Lecture Notes in Computer Science*, pages 393–403. Springer, 1995.

ST04. B. Schoenmakers and P. Tuyls. Practical Two-Party Computation Based on the Conditional Gate. In *Proceedings of ASIACRYPT '04*, volume 3329 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2004.

ST06. B. Schoenmakers and P. Tuyls. Efficient Binary Conversion for Paillier Encrypted Values. In *Proceedings of EUROCRYPT '06*, volume 4004 of *Lecture Notes in Computer Science*, pages 522–537. Springer, 2006.

Unr07. D. Unruh. Random Oracles and Auxiliary Input. In *Proceedings of CRYPTO '07*, volume 4622 of *Lecture Notes in Computer Science*, pages 205–223. Springer, 2007.

WB13. M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near-practicality. *Electronic Colloquium on Computational Complexity*, 20:165, 2013.

Wee09. H. Wee. Zero Knowledge in the Random Oracle Model, Revisited. In *Proceedings of ASIACRYPT '09*, volume 5912 of *Lecture Notes in Computer Science*, pages 417–434. Springer, 2009.

A Σ-Protocols for Multiplication and Decryption and Homomorphisms

A Σ-protocol proving correct multiplication due to [CDN01] is shown in Σ-protocol 2; a Σ-protocol proving correct decryption due to [Jur03] is shown in Σ-protocol 3.

We now exhibit homomorphisms for the Σ-protocols used in this paper. For the proof of plaintext knowledge Σ_{PK} (Σ-protocol 1), we combine proofs of

Σ-Protocol 2 Σ_{CM}: Proof of correct multiplication

```
[Relation]
    $R = \{((X,Y,Z),(y,r,s)) \mid Y = (1+N)^y r^N \wedge Z = X^y s^N\}$
[Announcement] $\Sigma.\mathsf{ann}((X,Y,Z),(y,r,s)) :=$
    $a \in_R \mathbb{Z}_N; u,v \in_R \mathbb{Z}_N^*; A := X^a v^N; B := (1+N)^a u^N;$ return $((A,B),(a,u,v))$
[Response] $\Sigma.\mathsf{res}((X,Y,Z),(y,r,s),(A,B),(a,u,v),c) :=$
    $t := \lfloor (a+cy)/N \rfloor; d := a+cy;\ e := ur^c(1+N)^t; f := vX^t s^c$
    return $(d,e,f)$
[Simulator] $\Sigma.\mathsf{sim}((X,Y,Z),c) :=$
    $d \in_R \mathbb{Z}_N; e,f \in_R \mathbb{Z}_N^*; A := X^d f^N Z^{-c}; B := (1+N)^d e^N Y^{-c}$
    return $((A,B),c,(d,e,f))$
[Extractor] $\Sigma.\mathsf{ext}((X,Y,Z),(A,B),c,c',(d,e,f),(d',e',f')) :=$
    $\beta, m := \langle$values such that $\beta(c-c') - mN = 1\rangle$
    return $((d-d')\beta, (e/e')^\beta Y^{-m}, (f/f')^\beta Z^{-m})$
[Verification]
    $\Sigma.\mathsf{ver}((X,Y,Z),(A,B),c,(d,e,f)) := (1+N)^d e^N \overset{?}{=} BY^c \wedge X^d f^N \overset{?}{=} AZ^c$
```

Σ-Protocol 3 Σ_{CD}: Proof of correct decryption

```
[Relation]
    $R = \{((d,d_i,v,v_i),\Delta s_i) \mid d_i^2 = d^{4\Delta s_i} \wedge v_i = v^{\Delta s_i}\}$
[Announcement] $\Sigma.\mathsf{ann}((d,d_i,v,v_i),\Delta s_i) :=$  // $k = \log_2 N$; $k_2$ stat. sec. param
    $u \in_R [0, 2^{2k+2k_2}]; a := d^{4u}; b := v^u;$ return $((a,b),r)$
[Response] $\Sigma.\mathsf{res}((d,d_i,v,v_i),\Delta s_i,(a,b),u,c) :=$
    $r := u + c\Delta s_i;$ return $r$
[Simulator] $\Sigma.\mathsf{sim}((d,d_i,v,v_i),c) :=$
    $z \in_R [0, 2^{2k+2k_2}];$ return$((d^{4z}(d_i)^{-2c}, v^z(v_i)^{-e}),c,z)$
[Extractor] $\Sigma.\mathsf{ext}((d,d_i,v,v_i),(a,b),c,c',r,r') :=$
    return $(r-r')/(c-c')$
[Verification] $\Sigma.\mathsf{ver}((d,d_i,v,v_i),(a,b),c,r) :=\ d^{4r} \overset{?}{=} a(d_i)^{2c} \wedge v^r \overset{?}{=} b(v_i)^c$
```

knowledge of the plaintexts x_i of $\{X_i\}$ into a proof of knowledge of the plaintext $\sum x_i$ of $\prod X_i$. Namely, let $\Phi.\mathsf{stmt}(\{X_i\}_{i\in I}) = \prod_{i\in I} X_i$ and $\Phi.\mathsf{ann}(\{B_i\}_{i\in I}) = \prod_{i\in I} B_i$. We would like to take $(d,e) = \Phi.\mathsf{resp}(\{(d_i,e_i)\}_{i\in I}) = (\sum_{i\in I} d_i, \prod_{i\in I} e_i)$, but because $\sum_{i\in I} d_i$ is computed modulo N, we need to add a correction factor $(1+N)^k$ ($k = \lfloor (\sum_{i\in I} d_i)/N \rfloor$) to e, i.e., $e = (\prod_{i\in I} e_i)(1+N)^k$. This homomorphism is defined for any collection of statements from V; it is easy to check the combination and randomness properties. (This homomorphism is not used in our protocols, but we give it for completeness.)

The proof of correct multiplication Σ_{CM} (Σ-protocol 2) has a homomorphism similar to that of the proof of knowledge. It is defined on statements $\{(X,Y_i,Z_i)\}_{i\in I}$ which share encryption X, and it proves that the multiplication under encryption of X with $\prod Y_i$ is equal to $\prod Z_i$. Formally, we let:

$$\Phi.\mathsf{stmt}(\{(X,Y_i,Z_i)\}_{i\in I}) = \left(X, \prod_{i\in I} Y_i, \prod_{i\in I} Z_i\right);$$

$$\Phi.\mathsf{ann}(\{A_i, B_i\}_{i \in I}) = \left(\prod_{i \in I} A_i, \prod_{i \in I} B_i\right);$$

$$\Phi.\mathsf{resp}(\{(d_i, e_i, f_i)\}_{i \in I}) = \left(\sum_{i \in I} d_i, \left(\prod_{i \in I} e_i\right)(1+N)^k, \left(\prod_{i \in I} f_i\right) Y^k\right),$$

with $k = \lfloor (\sum_{i \in I} d_i)/N \rfloor$.

Finally, proofs of correct decryption Σ_{CD} (Σ-protocol 3) can be combined into a proof of decryption with respect to an overall verification value. Let $I \geq \lceil n/2 \rceil$ be sufficiently many parties to decrypt a ciphertext, let $\{\lambda_i\}_{i \in I}$ be Lagrange interpolation coefficients for these parties[5], and let s_i be their shares of the decryption key $d = \sum_{i \in I} \Delta\lambda_i s_i$. Recall that decryption works by letting each party $i \in I$ provide decryption share $c_i = c^{2\Delta s_i}$; computing $c' = \prod_{i \in I} c_i^{2\Delta\lambda_i}$; and from this determining the plaintext as $\mathsf{paillierdecode}(c')$. Parties prove correctness of their decryption shares c_i by proving that $\log_{c^4} c_i^2 = \log_v v_i$, where v, v_i are publicly known verification values such that $v_i = v^{\Delta s_i}$. Now, if $\log_{c^4} c_i^2 = \log_v v_i$ for all i, then

$$\log_{c^4} c' = \log_{c^4} \prod_{i \in I} c_i^{2\Delta\lambda_i} = \log_v \prod_{i \in I} v_i^{\Delta\lambda_i} = \log_v \prod_{i \in I} (v^{\Delta s_i})^{\Delta\lambda_i} = \log_v v^{\Delta^2 d}.$$

Hence, decryption proofs for shares c_i with respect to verification values v_i can be combined into a decryption proof for c' with respect to verification value $v_0 := v^{\Delta^2 d}$. Formally, we let

$$\Phi.\mathsf{stmt}(\{(d, d_i, v, v_i)\}_{i \in I} = \left(d, \prod_{i \in I} c_i^{\Delta\lambda_i}, v, \prod_{i \in I} v_i^{\Delta\lambda_i}\right);$$

$$\Phi.\mathsf{ann}(\{(a_i, b_i)\}_{i \in I}) = \left(\prod_{i \in I} a_i^{\Delta\lambda_i}, \prod_{i \in I} b_i^{\Delta\lambda_i}\right);$$

$$\Phi.\mathsf{resp}(\{r_i\}_{i \in I}) = \sum \Delta\lambda_i r_i.$$

For the combination property of Definition 2, note that we really need $I \geq \lceil n/2 \rceil$ in order to apply Lagrange interpolation. For the randomness property, note that if $|C| < \lceil n/2 \rceil$, then at least one party in $I \notin C$ has a non-zero interpolation coefficient, hence the contribution of this party to the announcement ensures that the two combined announcements are different.

B Simulation-Based Security in the Random Oracle Model

We prove our protocols secure using the simulation paradigm in the random oracle model [BR93,Wee09]. In the random oracle model, evaluations of hash

[5] Note that λ_i are not always integral; but we will always use $\Delta\lambda_i$, which *are* integral.

function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{2l}$ are modelled as queries to a "random oracle" $\mathcal{O}$ that evaluates a perfectly random function. When simulating an adversary that operates in the random oracle model, the simulator also simulates the random oracle with respect to the adversary. In particular, it can choose how to respond to the adversary's queries (but, to achieve security, it should provide random values so that the adversary cannot distinguish between the real world and the simulation based on the output of the random oracle).

More precisely, we work in the explicitly programmable random oracle model without dependent auxiliary input [Wee09]. The random oracle is seen as a partial function that initially has an empty codomain (i.e., it is "without dependent auxiliary input"). In a real-world execution in this model, both the honest parties and the adversary use the random oracle for hash function evaluations. Namely, when a party calls the oracle on a value $v \in \text{dom}(\mathcal{O})$, it receives $\mathcal{O}(v)$; otherwise, a fresh random value is generated and $\mathcal{O}$ is updated accordingly. At the end of the execution, $\mathcal{O}$ contains all pairs of oracle queries made during the execution and their responses. In an ideal-world execution, the simulator can directly modify the preimage/image pairs in $\mathcal{O}$; the simulated adversary only has oracle access to $\mathcal{O}$ as in the real-world execution. Again, at the end of the simulation, $\mathcal{O}$ contains all values on which the oracle has been set. Computational (or statistical) indistinguishability between real and simulated executions is defined [Wee09] by stating that no PPT (or unbounded) algorithm can distinguish them, where the distinguisher has oracle access to $\mathcal{O}$. We prove slightly stronger versions of indistinguishability; namely, instead of giving the distinguisher oracle access to $\mathcal{O}$, we simply supply it with the full list $\mathcal{O}$. We can then simply use the normal, non-oracle, definitions for indistinguishability; this is clearly at least as strong.

Our proofs rely on the absence of dependent auxiliary input. When a party presents a non-interactive proof of knowledge, we perform rewinding to find the witness to that proof; but if oracle queries before the protocol execution are allowed, then a party may replay a proof that was performed before the protocol execution, making rewinding impossible. As noted in [Unr07], this suggests that the random oracle in our protocols should be instantiated with a keyed hash function, where every protocol instance uses a different key. See Section 5 for a discussion.

Figure 2 lists the notation we use when presenting simulators in the random oracle model. We use global variables $\mathcal{A}$ and $\mathcal{O}$ to denote the current state of the adversary and the random oracle. An invocation of $\mathcal{A}$ with oracle access to $\mathcal{O}$ is denoted $v := \mathcal{A}^{\mathcal{O}}(w)$; afterwards, both $\mathcal{O}$ and $\mathcal{A}$ are updated to reflect the respective new states.

B.1 The Fiat-Shamir Heuristic and Witness-Extended Emulation

When the Fiat-Shamir heuristic (Protocol 1) is used, in some situations a simulator can extract witnesses from proofs by corrupted parties. Namely, in [Gro04], Groth showed that, by simulating an adversary using *witness-extended emulation*, a simulator can obtain witnesses for all proofs that the adversary produces. Specifically, Groth proved the following:

$\mathrm{dom}(\mathcal{O}), \mathrm{codom}(\mathcal{O}), \mathrm{rng}(\mathcal{O})$	Domain/codomain/range of random oracle $\mathcal{O}$, seen as partial function
fail	Terminate simulation, returning special error value
$\mathcal{A}, \mathcal{O}$	Global variable: simulated attacker/random oracle
$v := \mathcal{A}^{\mathcal{O}}(w)$	Exchange values v, w with $\mathcal{A}$ having oracle access to $\mathcal{O}$

Fig. 2. Notation and conventions for simulation-based security in the Random Oracle Model

Theorem 2 ([Gro04]). *Let $(\mathcal{A}', \mathcal{O}', \boldsymbol{x}, \boldsymbol{p}) \leftarrow \mathcal{A}^{\mathcal{O}}(z)$ be an adversary $\mathcal{A}$ interacting with random oracle $\mathcal{O}$ that, on some polynomial-length input z, outputs a list $\boldsymbol{x}$ of statements, and a list $\boldsymbol{p}$ of corresponding validating Fiat-Shamir proofs. Then there exists a PPT emulator $(\mathcal{A}', \mathcal{O}', \boldsymbol{x}, \boldsymbol{p}, \boldsymbol{w}) \leftarrow E_{\mathcal{A}}^{\mathcal{O}}(z)$ such that the part $(\mathcal{A}', \mathcal{O}', \boldsymbol{x}, \boldsymbol{p})$ of the output of $E_{\mathcal{A}}$ is perfectly indistinguishable from the output of $\mathcal{A}$, and $\boldsymbol{w}$ are witnesses corresponding to the statements $\boldsymbol{x}$.*

Essentially, the witness-extended emulator $E_{\mathcal{A}}$ simulates the adversary $\mathcal{A}$, keeping track of all oracle queries it makes. For each valid proof that $\mathcal{A}$ produces, it rewinds $\mathcal{A}$ to the point of the oracle query used to obtain the challenge, and keeps on reprogramming the random oracle until $\mathcal{A}$ again produces a correct proof with a new challenge. It finally extracts the witness using the special soundness property of the Σ-protocol.

However, a major limitation of the technique of [Gro04] is that it only considers a single invocation of the adversary. On the other hand, the CDN protocol consists of different rounds in which the adversary is invoked with inputs from the honest parties. Now, if the adversary provides a proof in its rth invocation, it may have already queried the random oracle for the announcement of that proof in an earlier invocation $s < r$. Then, to extract the witness for the proof from invocation r, we need to simulate the adversary from invocation s. This means that we also need to re-compute the messages from the honest party to the adversary for all rounds between s and r. However, computing these messages in general requires witnesses for the proofs of the adversary between rounds s and r. Hence, to extract the witnesses for the rth invocation, we need to recursively extract witnesses for all rounds between s and r, which in turn may also require recursive rewinding. Hence, if Fiat-Shamir proofs take place in R invocations of the adversary, then witness-extended emulation may increase the running time of a simulator by a factor $O(R!)$. Because we need simulators to be PPT, $R!$ should be polynomial in the security parameter, so we can use the Fiat-Shamir heuristic, but only in essentially a constant number of rounds (as in our UVCDN protocol, where we only use the normal Fiat-Shamir heuristic at the beginning and the end of the protocol).

Our multiparty Fiat-Shamir heuristic (Section 3.2) addresses the above limitations while also combining proofs of individual parties into one single proof for a combined statement. If this combination is not desired, then each party can simply hash the concatenation of all parties' announcements instead of taking combinations $\Phi.\mathsf{stmt}(\{v_i\}_{i \in P \setminus F}), \Phi.\mathsf{ann}(\{a_i\}_{i \in P \setminus F}$. Note that also in this case, the extra round in which parties commit to their announcements is still needed.

Without this round, the adversary could choose its announcement after the honest parties chose theirs, and hence, the simulator would not know on which preimage to program the random oracle when simulating the honest parties' proofs. In the non-combination case, the parties could alternatively broadcast their announcements; and then use a hash of the concatenation of the announcements and some fresh randomness. This way, the simulator can simulate the honest parties' proofs because the adversary cannot predict the preimage before seeing the proof; while the adversary cannot make oracle queries for its own proofs too early because it needs to wait for the honest parties' announcements. Another way of addressing the limitations in the non-combination case would be to add the full previous communication transcript to the hash (or, in any case, some recent messages that contain sufficient entropy by honest parties): this way, rewinding is guaranteed to be to a recent moment in the protocol.

C Simulator of the Multiparty Fiat-Shamir Heuristic

Our simulator $\mathcal{S}_{\mathrm{M}\Sigma}$ of the multiparty Fiat-Shamir heuristic is shown in Algorithm 2. We now explain the general idea, deferring the discussion of exceptional cases to the security proof.

As long as flag stage2 is not set, the simulator behaves like the protocol. First, the parties exchange hashed commitments (line 10–12). Namely, the simulator generates random hash values for the honest parties (line 10), and receives hash values for the corrupted parties (line 11). Note that the adversary can later open these values only if they come from the random oracle, hence the simulator knows the adversary's pre-images a_i (line 12). Next, the parties open their commitments (lines 16–26). The simulator generates a random challenge c (line 16); simulates the Σ-protocol for the honest parties (line 18); and programs the random oracle so that the announcements of the honest parties hash to the values h_i supplied earlier (lines 19–20). If possible, the simulator combines the announcements a_i of the honest and corrupted parties, and programs the random oracle to return c on the result (lines 21–24). The simulator then the receives announcements a'_i for the corrupted parties (line 25); and checks if they are correct (line 26); if so, it exchanges responses to the challenge (line 29). If all parties provided correct responses, then the protocol terminates: in this case, the simulator stores the state at this point for returning it when the simulation ends, and sets flag stage2 (line 33). Otherwise, the simulator continues repeats the above process with the parties that have not misbehaved so far (line 40).

When flag stage2 is set, the simulator has simulated one successful run of the multiparty Fiat-Shamir heuristic for which it will now extract witnesses. Note that variables $\mathcal{A}_{\mathrm{start}}$, $\mathcal{O}_{\mathrm{start}}$, and F' contain the state of the protocol at the point when the adversary has supplied hash values h_i for which the simulator knows the pre-images a_i, and a challenge c' with correct responses $\{r'_i\}_{i \in (P \setminus F') \cap C}$. To extract witnesses for the parties in $(P \setminus F') \cap C$, the simulator keeps re-winding to state $(\mathcal{A}_{\mathrm{start}}, \mathcal{O}_{\mathrm{start}}, F')$ (line 15), and repeats the above simulation procedure (lines 16–30), until the adversary has again produced correct responses for the same parties (line 31). In this case, it has responses from the adversary for the

Algorithm 2 $\mathcal{S}_{\mathrm{M}\Sigma}$: simulator for Multiparty Fiat-Shamir Heuristic

```
1.  // pre: $\Sigma$: $\Sigma$-protocol with homomorphism. $\Phi$; $P$: non-failed parties ($P \cap F = \emptyset$);
2.  //         $v_P = \{v_i\}_{i \in P}$: statements
3.  // post: if $|P \setminus F| \geq \lceil n/2 \rceil$, then $v_{P \setminus F}$ is the statement $\Phi.\mathsf{stmt}(\{v_i\}_{i \in P \setminus F})$, $\pi_{P \setminus F}$
4.  //          is corresponding FS proof, and $\forall i \in (P \setminus F) \cap C$, $w_i$ is a witness for $v_i$
5.  // invariant: $F \subset C$: set of failed parties only includes corrupted parties
6.  $(v_{P \setminus F}, \pi_{P \setminus F}, w_{(P \setminus F) \cap C}) \leftarrow \mathcal{S}_{\mathrm{M}\Sigma}(\Sigma, \Phi, P, v_P, aux = ") :=$
7.    $\mathsf{stage2} := \perp$
8.    repeat                                                          // line 7 of M$\Sigma$
9.      if $\mathsf{stage2} = \perp$ then      // stage 1 of simulation: simulate protocol w.r.t. $\mathcal{A}$
10.       foreach $i \in P \setminus C$ do $h_i \in_R \mathrm{codom}(\mathcal{O})$
11.       $(\mathcal{A}, \mathcal{O}, \{h_i\}_{i \in (P \setminus F) \cap C}) := \mathcal{A}^{\mathcal{O}}(\{h_i\}_{i \in P \setminus C})$          // line 8 of M$\Sigma$
12.       foreach $i \in (P \setminus F) \cap C$ do $a_i := (x$ if $\exists! x : (x||i, h_i) \in \mathcal{O}$ else $\perp)$
13.       $\mathcal{A}_{\mathrm{start}} := \mathcal{A}; \mathcal{O}_{\mathrm{start}} := \mathcal{O}$
14.     else                                 // stage 2: extract witnesses from $\mathcal{A}_{\mathrm{start}}$
15.       $\mathcal{A} := \mathcal{A}_{\mathrm{start}}; \mathcal{O} := \mathcal{O}_{\mathrm{start}}; F := F'$
16.     $c \in_R \mathrm{codom}(\mathcal{O})$
17.     foreach $i \in P \setminus C$ do
18.       $(a_i, r_i) := \Sigma.\mathsf{sim}(v_i, c)$
19.       if $a_i||i \in \mathrm{dom}(\mathcal{O})$ then fail
20.       $\mathcal{O} := \mathcal{O} \cup \{(a_i||i, h_i)\}$
21.     if $\forall i \in (P \setminus F) \cap C : a_i \neq \perp$ then    // know combined ann.: program oracle
22.       $a := \Phi.\mathsf{stmt}(\{v_i\}_{i \in P \setminus F})||\Phi.\mathsf{ann}(\{a_i\}_{i \in P \setminus F})||aux$        // line 12 of M$\Sigma$
23.       if $a \in \mathrm{dom}(\mathcal{O})$ then fail
24.       $\mathcal{O} := \mathcal{O} \cup \{(a, c)\}$
25.     $\{a'_i\}_{i \in (P \setminus F) \cap C}) := \mathcal{A}^{\mathcal{O}}(\{a_i\}_{i \in P \setminus C})$                       // line 9 of M$\Sigma$
26.     $F' := F; F := F \cup \{i \in (P \setminus F) \cap C \mid (a'_i||i, h_i) \notin \mathcal{O}\}$        // line 10 of M$\Sigma$
27.     if $F = F'$ then                                              // line 11 of M$\Sigma$
28.       if $\exists i \in (P \setminus F) \cap C : a_i = \perp \vee a_i \neq a'_i$ then fail
29.       $\{r_i\}_{i \in (P \setminus F) \cap C} := \mathcal{A}^{\mathcal{O}}(\{r_i\}_{i \in P \setminus C})$                    // line 13 of M$\Sigma$
30.       $F := F \cup \{i \in P \setminus F \mid \neg \Sigma.\mathsf{ver}(v_i, a_i, c, r_i)\}$                 // line 14 of M$\Sigma$
31.       if $F = F'$ then                                            // line 15 of M$\Sigma$
32.         if $\mathsf{stage2} = \perp$ then           // end of stage 1: save adversary state
33.           $\mathcal{A}_{\mathrm{ret}} := \mathcal{A}; \mathcal{O}_{\mathrm{ret}} := \mathcal{O}; c' := c; \{r'_i\}_{i \in (P \setminus F) \cap C} := \{r_i\}_{i \in (P \setminus F) \cap C}$
34.           $\pi_{\mathrm{ret}} := (\Phi.\mathsf{ann}(\{a_i\}_{i \in P \setminus F}), c, \Phi.\mathsf{resp}(\{r_i\}_{i \in P \setminus F})); \mathsf{stage2} := \top$
35.         else
36.           if $c = c'$ then fail
37.           foreach $i \in (P \setminus F) \cap C$ do $w_i := \Sigma.\mathsf{ext}(v_i, a_i, c, c', r_i, r'_i)$
38.           $\mathcal{A} := \mathcal{A}_{\mathrm{ret}}; \mathcal{O} := \mathcal{O}_{\mathrm{ret}}$
39.           return $(\Phi.\mathsf{stmt}(\{v_i\}_{i \in P \setminus F}), \pi_{\mathrm{ret}}, \{w_i\}_{i \in (P \setminus F) \cap C})$  // line 16 of M$\Sigma$
40.   until $\mathsf{stage2} = \perp \wedge |P \setminus F| < \lceil n/2 \rceil$                           // line 17 of M$\Sigma$
41.   return $(\perp, \perp, \emptyset)$                                             // line 18 of M$\Sigma$
```

same announcement with different challenges, from which it can extract witnesses using the special soundness property of the Σ-protocol (line 37). Finally, the simulator returns the adversary state after the first successful protocol run, along with the proof and the witnesses for the contributing corrupted parties (line 39).

We now prove that simulator $\mathcal{S}_{\mathrm{M}\Sigma}$ satisfies soundness and zero-knowledge. The statement of our lemma is analogous to the statement from [CDN01], with two technical differences. First, we do not use trapdoor commitments, in effect replacing them by the random oracle. Second, we do not guarantee perfect indistinguishability because our simulator may occasionally fail. We get the following:

Lemma 1. *Assume that fewer than $\lceil n/2 \rceil$ parties are corrupted. Define real-world executions of $M\Sigma$ and ideal-world executions of $\mathcal{S}_{M\Sigma}$ as follows:*

- *Let $(\mathcal{A}', \mathcal{O}', F', v_{P \setminus F'}, \pi_{P \setminus F'}) \leftarrow exec_{M\Sigma}(\mathcal{A}, \mathcal{O}, F, \Sigma, \Phi, P, v_P, w_P, aux)$ denote a run of the $M\Sigma$ protocol (Protocol 2) with initial adversary state $\mathcal{A}$, random oracle $\mathcal{O}$, set F of failed parties, parameters $(\Sigma, \Phi, P, v_P, w_P, aux)$, final states $\mathcal{A}', \mathcal{O}', F$, and return values $v_{P \setminus F}, \pi_{P \setminus F}$.*
- *Let $(\mathcal{A}', \mathcal{O}', F', v_{P \setminus F'}, \pi_{P \setminus F'}, w_{(P \setminus F) \cap C}) \leftarrow sim_{\mathcal{S}_{M\Sigma}}(\mathcal{A}, \mathcal{O}, F, \Sigma, \Phi, P, v_P, aux)$ denote a run of the $\mathcal{S}_{M\Sigma}$ simulator with initial adversary state $\mathcal{A}$, random oracle state $\mathcal{O}$, and set F of failed parties, and parameters $(\Sigma, \Phi, P, v_P, aux)$; and final states $\mathcal{A}', \mathcal{O}', F'$ and return values $v_{P \setminus F'}, \pi_{P \setminus F'}, w_{(P \setminus F') \cap C}$.*

Then $\mathcal{S}_{M\Sigma}$ is a PPT algorithm satisfying the following two properties:

Soundness *Except with negligible probability, $w_{(P \setminus F') \cap C} = \{w_i\}_{i \in (P \setminus F') \cap C}$ are valid witnesses for the statements v_i of the corrupted parties in P that produced verifying proofs.*

Zero-Knowledge *The part $(\mathcal{A}', \mathcal{O}', F', v_{P \setminus F'}, \pi_{P \setminus F'})$ of the output of $sim_{\mathcal{S}_{M\Sigma}}$ is statistically indistinguishable from the output of $exec_{M\Sigma}$.*

Proof. We need to show that $\mathcal{S}_{\mathrm{M}\Sigma}$ is PPT, and that it satisfies soundness and zero-knowledge.

First, we show that the simulator only fails with negligible probability: It fails in line 19 if the random oracle has already been programmed on a simulated announcement. This happens with negligible probability because of the non-triviality property of the Σ-protocol. It fails in line 23 if the random oracle has already been programmed on a homomorphically combined announcement. This happens with negligible probability because of the randomness property of the homomorphism Φ. It fails in line 28 if the adversary manages to supply a pre-image a'_i different from the one calculated in line 12. Then the adversary has found a collision or it has found a pre-image of h_i without getting h_i from the oracle, which happens with negligible probability. Finally, it fails in line 36 if the simulator has twice generated the same challenge randomly in the codomain of the random oracle. But this codomain is $\{0,1\}^{2l}$ with l a security parameter, so also this happens with negligible probability.

For soundness: the adversary returns values $w_i = \Sigma.\mathsf{ext}(v_i, a_i, c, c', r_i, r'_i)$ for which we know that $\Sigma.\mathsf{ver}(v_i, a_i, c, r_i)$ and $\Sigma.\mathsf{ver}(v_i, a_i, c', r'_i)$ hold (by line 30), and $c \neq c'$ (by line 36). Hence, the w_i are valid witnesses by the special soundness property of the Σ-protocol.

For zero-knowledge, note that the part $(\mathcal{A}', \mathcal{O}', F', v_{P \setminus F'}, \pi_{P \setminus F'})$ returned by the simulated execution is determined while $\mathsf{stage2} = \bot$. Namely, one checks that $\mathcal{A}'$ and $\mathcal{O}'$ are $\mathcal{A}_{\mathrm{ret}}$ and $\mathcal{O}_{\mathrm{ret}}$ as set in line 33; F' is the value F when line 33 was executed; $v_{P\setminus F'}$ depends only on that value F'; and $\pi_{P\setminus F'}$ is set in line 34. But observe that, while $\mathsf{stage2} = \perp$, the simulator behaves like the original protocol with respect to the adversary, except in three ways. The first is that the protocol may fail, but as shown, this happens only with negligible probability so it does not affect statistical indistinguishability. The second is that, in line 26 of the simulator, we check if $(a'_i||i, h_i) \in \mathcal{O}$ instead of querying the random oracle. In the protocol, it might be that $a'_i||i$ was not queried before, but when queried, the random oracle happens to return h_i. Again, this happens with only negligible probability. Finally, all proofs of the honest parties are simulated. The special honest-verifier zero-knowledge property of the Σ-protocol now implies that $(\mathcal{A}', \mathcal{O}', F', v_{P\setminus F'}, \pi_{P\setminus F'})$ is identically distributed in the real and simulated protocol run.

Finally, we need to show that the simulator runs in PPT. For this, let us analyse how often the main **repeat** . . . **until** loop (lines 8–40) is executed. If $\mathsf{stage2} = \perp$ at the beginning of the loop, then either $|P \setminus F|$ becomes strictly smaller, or $\mathsf{stage2}$ is set to $\top$. Hence, the time spent while $\mathsf{stage2} = \perp$ is certainly polynomial. Now, consider the loop execution in which $\mathsf{stage2}$ is set to $\top$, i.e., in which all provers provide correct announcements and responses for the first time. Let ϵ be the a priori probability, from line 16 of $\mathcal{S}_{\mathrm{M}\Sigma}$, that all provers indeed provide correct announcements and responses. Then with probability ϵ, line 32 is reached; and afterwards, the loop is executed from the same state (line 15) until we again get correct announcements and proofs, which takes an expected $1/\epsilon$ number of tries. Hence, the loop is executed with $\mathsf{stage2} = \top$ with probability ϵ for an expected $1/\epsilon$ number of times, hence it contributes polynomially to the running time. This completes the proof. □

Note that the above proof relies on the fact that communication happens on a broadcast channel. Namely, in this case, all honest parties simulate their Σ-protocol based on the same challenge. However, we remark that the simulator could also be made to work in a non-broadcast setting. In this case, in line 12, it determines announcements of the corrupted parties with respect to each honest party; and based on that, decides for how many different challenges to simulate the honest parties' Σ-protocols.

D Security of the UVCDN Protocol: Proof of Theorem 1

To prove security of the UVCDN protocol, we need to build a simulator for every adversary $\mathcal{A}$. Because the honest majority and corrupted majority cases are very different, we define two simulators: simulator $\mathcal{S}^{\mathrm{honest}}_{\mathrm{UVCDN}}$ (ITM 2) for the honest majority case; and simulator $\mathcal{S}^{\mathrm{corrupt}}_{\mathrm{UVCDN}}$ (ITM 3) for the corrupted majority case. Below, we present the simulators and prove that they correctly simulate the protocol in their respective cases. Theorem 1 follows directly from Lemmas 2 and 3 below.

ITM 2 $\mathcal{S}^{\text{honest}}_{\text{UVCDN}}$: Simulator for Universally Verifiable CDN (honest majority)

```
// pre: $\mathsf{pk}$ public key; $C$ ($|C| < \lceil n/2 \rceil$) corrupted parties with secret keys
//          $\{s_i\}_{i \in \mathcal{P} \cap C}$, inputs $\{x_i\}_{i \in \mathcal{I} \cap C}$
// post: attacker simulated, values provided to $\mathcal{T}$ to mimic real execution outputs
$(r, \mathcal{O}) \leftarrow \mathcal{S}^{\text{honest}}_{\text{UVCDN},\mathcal{A}}(\mathsf{pk}{=}(N, v, v_0, \{v_i\}_{i \in \mathcal{P}}), C, \{x_i\}_{i \in \mathcal{I} \cap C}, \{s_i\}_{i \in \mathcal{P} \cap C}, a) :=$
    $F := \emptyset$                                            // parties that have misbehaved
    // input phase: input parties provide inputs and prove knowledge of them
    foreach $i \in \mathcal{I} \setminus C$ do $x_i := 0; r_i \in_R \mathbb{Z}_N^*; X_i := (1+N)^{x_i} r_i^N$
    $\{X_i\}_{i \in \mathcal{I} \cap C} := \mathcal{A}^{\mathcal{O}}(\{X_i\}_{i \in \mathcal{I} \setminus C})$
    $(\_, \{(x_i, \_)\}_{i \in (\mathcal{I} \setminus F) \cap C}) := \mathcal{S}_{\text{F}\Sigma}(\Sigma_{\text{PK}}, \mathcal{I}, \{X_i\}_{i \in \mathcal{I}})$
    foreach $i \in \mathcal{I} \cap F$ do $x_i := 0;\ X_i := 1$
    $\mathsf{send}(\{x_i\}_{i \in \mathcal{I} \cap C}, \mathcal{T})$
    // computation phase: go through the circuit gate-by-gate
    foreach gate do
        if ⟨constant gate $c$ with value $v$⟩ then $x_c := v;\ X_c := (1+N)^v$
        if ⟨addition gate $c$ with inputs $a$, $b$⟩ then $x_c := x_a + x_b; X_c := X_a X_b$
        if ⟨subtraction gate $c$ with inputs $a$, $b$⟩ then $x_c := x_a - x_b; X_c := X_a X_b^{-1}$
        if ⟨multiplication gate $c$ with inputs $a$, $b$⟩ then       // [DN03] multiplication
            foreach $i \in \mathcal{P} \setminus C$ do
                $d_i \in_R \mathbb{Z}_N; r_i, t_i \in_R \mathbb{Z}_N^*;\ D_i := (1+N)^{d_i} r_i^N; E_i := (X_b)^{d_i} t_i^N$
            $x_c := x_a x_b; m \in_R \mathcal{P} \setminus C; D_m := D_m X_a^{-1}; E_m := E_m (1+N)^{-x_c}$
            $\{(D_i, E_i)\}_{i \in (\mathcal{P} \setminus F) \cap C} := \mathcal{A}^{\mathcal{O}}(\{(D_i, E_i)\}_{i \in \mathcal{P} \setminus C})$
            $((\_, D_c, E_c), \pi_{\text{CM}c}, \{(d_i, \_, \_)\}_{i \in (\mathcal{P} \setminus F) \cap C}) :=$
                    $\mathcal{S}_{\text{M}\Sigma}(\Sigma_{\text{CM}}, \Phi_{\text{CM}}, \mathcal{P} \setminus F, \{(X_b, D_i, E_i)\}_{i \in \mathcal{P} \setminus F})$
            $S_c := X_a \cdot D_c; \{S_i\}_{i \in \mathcal{P} \setminus C} := \langle$decr. shares s.t. $S_c$ decrypts to $\Sigma_{i \in \mathcal{P} \setminus F} d_i\rangle$
            $\{S_i\}_{i \in (\mathcal{P} \setminus F) \cap C} := \mathcal{A}^{\mathcal{O}}(\{S_i\}_{i \in \mathcal{P} \setminus C})$
            $((\_, S_{0,c}, \_, \_), \pi_{\text{CD}c}, \_) := \mathcal{S}_{\text{M}\Sigma}(\Sigma_{\text{CD}}, \Phi_{\text{CD}}, \mathcal{P} \setminus F, \{(S_c, S_i, v, v_i)\}_{i \in \mathcal{P} \setminus F})$
            $X_c := (X_b)^{\Sigma_{i \in \mathcal{P} \setminus F} d_i} E_c^{-1}$
    // output phase: decrypt result for result party
    if $\mathcal{R} \notin C$ then $X := \mathsf{recv}(\mathcal{T}); y \in_R \mathbb{Z}_N; D := (1+N)^{-y} X; \mathcal{A}^{\mathcal{O}}(D)$
    if $\mathcal{R} \in C$ then $r := \mathsf{recv}(\mathcal{T}); D := \mathcal{A}^{\mathcal{O}}()$
    $(\pi_{\text{PK}d}, \{(d_i, \_)\}_{i \in \{\mathcal{R}\} \cap C}) := \mathcal{S}_{\text{F}\Sigma}(\Sigma_{\text{PK}}, \{\mathcal{R}\}, \{D\})$
    if $\mathcal{R} \notin F$ then
        if $\mathcal{R} \in C$ then $y := r - d_{\mathcal{R}}$
        $Y := X_{\text{outgate}} \cdot D^{-1}; \{Y_i\}_{i \in \mathcal{P} \setminus C} := \langle$decr. shares s.t. $Y$ decrypts to $y\rangle$
        $\{Y_i\}_{i \in (\mathcal{P} \setminus F) \cap C} := \mathcal{A}^{\mathcal{O}}(\{Y_i\}_{i \in \mathcal{P} \setminus C})$
        $((\_, Y_0, \_, \_), \pi_{\text{CD}y}, \_) := \mathcal{S}_{\text{M}\Sigma}(\Sigma_{\text{CD}}, \Phi_{\text{CD}}, \mathcal{P} \setminus F, \{(Y, Y_i, v, v_i)\}_{i \in \mathcal{P} \setminus F}, D)$
        $\pi := (\{(D_c, E_c, \pi_{\text{CM}c}, S_{0,c}, \pi_{\text{CD}c})\}_{c \in \text{gates}}, (D, \pi_{\text{PK}d}, Y_0, \pi_{\text{CD}y}))$
    // proof phase: if computation succeeded, provide proof to verifier
    if $\mathcal{R} \in C$ then  // result party corrupted: tell $\mathcal{T}$ whether to provide proof to $\mathcal{V}$
        if $\mathcal{V} \notin C$ then $\pi := \mathcal{A}^{\mathcal{O}}()$
        if $\mathsf{vercomp}(\mathsf{pk}, \{X_i\}_{i \in \mathcal{I}}, \pi)$ then $\mathsf{send}(s, \mathcal{T})$ else send$(\perp, \mathcal{T})$
    else if $\mathcal{V} \in C$ then $\mathcal{A}^{\mathcal{O}}(\pi)$          // verifier corrupted: expects to receive proof
    return $(\mathcal{A}^{\mathcal{O}}(), \mathcal{O})$   // return adv.-chosen output and simulated random oracle
```

D.1 The Honest Majority Case

If fewer than $\lceil n/2 \rceil$ parties are corrupted, then we guarantee privacy like the original CDN protocol. Our simulator $\mathcal{S}^{\text{honest}}_{\text{UVCDN}}$ (ITM 2) for this case is an adaptation of CDN simulator from [CDN01]. Until line 27 of the simulator, it follows the CDN simulator except in two ways: it simulates the optimised [DN03] multiplication protocol, and it simulates non-interactive proofs rather than interactive proofs.

First, lines 18–27 of $\mathcal{S}^{\text{honest}}_{\text{UVCDN}}$ simulate the multiplication protocol from [DN03]. Analogously to Theorem 3 of [CDN01], lines 18–27 simulate multiplication given encryptions X_a, X_b of the values to be multiplied, and plaintext x_c of the result[6]. This simulation is constructed to be statistically indistinguishable from the multiplication protocol itself (lines 15–26 of UVCDN). The simulator generate values d_i, r_i, t_i, D_i, E_i on behalf of the honest parties as in the protocol (lines 18–19); but then multiplies one of the resulting (D_i, E_i) by X_a^{-1} and $(1+N)^{-x_c}$, respectively (line 20). This way, the simulator knows that $S_c = X_a \cdot D_c = X_a \cdot \prod_{i \in \mathcal{P} \setminus F} D_i$ computed in line 24 encrypts $\sum_{i \in \mathcal{P} \setminus F} d_i$. It simulates the proofs of correct multiplication of the (D_i, E_i) (line 22-23); and computes S_c as in the protocol (line 24). Finally, it simulates decryption of S_c to $\sum_{i \in \mathcal{P} \setminus F} d_i$ (lines 25–26), and computes X_c as in the protocol (line 27). One checks that this is indeed a statistically indistinguishable simulation.

Second, $\mathcal{S}^{\text{honest}}_{\text{UVCDN}}$ simulates the (multiparty) Fiat-Shamir proofs in a statistically indistinguishable way, while also giving witnesses for the corrupted parties. Recall from Section 3.1 that we can simulate the Fiat-Shamir heuristic provided that $R!$ is polynomial, where R is the number of rounds in which Fiat-Shamir proofs take place. In our case, $R = 2$ so simulation is possible. Slightly abusing notation, we denote the simulation of the Fiat-Shamir heuristic (which implicitly includes recursive calls to the simulator) as $(\pi_{P \setminus F}, w_{(P \setminus F) \cap C}) \leftarrow \mathcal{S}_{\text{F}\Sigma}(\Sigma, P, v_P)$. For the multiparty Fiat-Shamir heuristic, we use simulator $\mathcal{S}_{\text{M}\Sigma}$ (Algorithm 2) whose correctness we have proven in Appendix C.

Our simulation of the output and proof phases (line 28–43 of the simulator) differs from the simulation in [CDN01]. Namely, in our case, simulation of the output phase results in an encryption of the actual result of the computation (obtained from the trusted party), whereas the CDN simulator works with a simulated encryption until the end. Indeed, if the result party is honest, our simulator receives an encryption X of the actual result (line 29). Recall that the verifier sets the encrypted computation result to $(1+N)^y D$ with D provided by the result party, so we simulate D from the result party as $D = (1+N)^{-y} X$ to ensure that the verifier's output will be X. If the result party is corrupted, the simulator receives D from the adversary. To ensure that $(1+N)^y D$ indeed encrypts the result of the computation, the simulator extracts plaintext $d_{\mathcal{R}}$ of D (line 31), and threshold decrypts $X_{\text{outgate}} \cdot D^{-1}$ to $y = r - d_{\mathcal{R}}$ (line 33). Apart

[6] Technically, the multiplication simulator of [CDN01] uses an encryption of the result that it determined at the start of the CDN simulation, whereas we in effect determine this encryption during the gate-by-gate evaluation of the circuits. These two methods are clearly equivalent.

from this, the output phase is as in [CDN01]. Finally, in the proof phase, the simulator provides an honest proof to a corrupted verifier; or receives the proof of a corrupted result party for an honest verifier. In the latter case, if the verifier receives a correct proof, then the randomness used will be $s_{\mathcal{R}}$ (as we will show later), so the simulator provides this randomness to the trusted party.

Lemma 2. *For all inputs $x_1, \ldots, x_m, C, a$ with $|\mathcal{P} \cap C| < \lceil n/2 \rceil$, $\mathcal{S}^{honest}_{UVCDN,\mathcal{A}}$ is PPT, and*

$$EXEC_{UVCDN,\mathcal{A}}(k, (x_1, \ldots, x_m), C, a) \text{ and } IDEAL_{f,\mathcal{S}_{UVCDN,\mathcal{A}}}(k, (x_1, \ldots, x_m), C, a)$$

are computationally indistinguishable in security parameter k.

Proof. We remark that $\mathcal{S}^{\text{honest}}_{\text{UVCDN}}$ is clearly PPT since it only calls PPT subroutines.

For computation indistinguishability, we extend the proof for the CDN protocol from [CDN01]. In this proof, an algorithm $\text{YAD}_{\mathcal{A}}(B, k, (x_1, \ldots, x_m), C, a)$ is defined that, depending on an encrypted bit $B = (1+N)^b r^N$, simulates either an execution of the protocol with the real inputs (if B encrypts 1); or an execution with zero inputs for the honest parties (if B encrypts 0). It is shown that the $b = 0$ case is statistically indistinguishable to $\text{IDEAL}_{f,\mathcal{S}_{\mathcal{A}}}(k, (x_1, \ldots, x_m), C, a)$ and the $b = 1$ case is statistically indistinguishable to $\text{EXEC}_{\text{CDN},\mathcal{A}}(k, (x_1, \ldots, x_m), C, a)$. Moreover, the $b = 0$ and $b = 1$ cases are computationally indistinguishable by semantic security of the Paillier threshold cryptosystem (otherwise, the distinguisher between the two cases can distinguish random encryptions of 0 and 1).

To prove computational indistinguishability for the UVCDN protocol, we need to modify the algorithm $\text{YAD}_{\mathcal{A}}(B, k, (x_1, \ldots, x_m), C, a)$ to take into account the changed output of the result party and the additional output of the verifier. Namely, for honest $\mathcal{R}$, $\text{YAD}_{\mathcal{A}}$ outputs the result of the (real) computation and randomness s for encryption D from line 29 of the protocol; for honest $\mathcal{V}$, $\text{YAD}_{\mathcal{A}}$ outputs the return value of `vercomp`. Statistical indistinguishability of the $b = 1$ case to $\text{EXEC}_{\text{UVCDN},\mathcal{A}}$ and computational indistinguishability of the $b = 0$ and $b = 1$ cases are proven analogously to [CDN01]. For statistical indistinguishability of the $b = 0$ case to $\text{IDEAL}_{f,\mathcal{S}_{\text{UVCDN},\mathcal{A}}}$, the output of $\mathcal{V}$ in case of a corrupted $\mathcal{R}$ is problematic: in $\text{YAD}_{\mathcal{A}}$ it returns the output of `vercomp`, but in $\text{IDEAL}_{f,\mathcal{S}_{\text{UVCDN},\mathcal{A}}}$, it outputs $(1 + N)^r s^N$ with s the randomness of D in line 29 of UVCDN. These values coincide if $\mathcal{R}$ provides the same Y_0 and D to $\mathcal{V}$ that it computed with the computation parties during the protocol; but $\mathcal{R}$ could of course present a Y_0 or D with a valid proof to $\mathcal{V}$, in which case the output of $\mathcal{V}$ in the two distributions is different.

However, this latter possibility can only happen with negligible probability. For suppose that it happens with non-negligible probability. Because Y_0 and D are both input to the proof of correct decryption (D being the auxiliary input), $\mathcal{R}$ needs to provide a valid proof of correct decryption apart from the one produced in lines 34–35 of UVCDN. But from this proof, witness $\Delta^2 d$ can be extracted by witness-extended emulation. From this, we can build a distinguisher that, given an encryption B, runs $\text{YAD}_{\mathcal{A}}(r^N, k, (x_1, \ldots, x_m), C, a)$ (r random), and, whenever it gets witness $\Delta^2 d$, outputs the decryption of B. By the above discussion, this distinguisher succeeds with non-negligible probability, contradicting semantic security of the Paillier threshold cryptosystem. So, $\text{IDEAL}_{f,\mathcal{S}_{\text{UVCDN}},\mathcal{A}}$ and $\text{YAD}_{\mathcal{A}}$ with $b = 0$ are in fact statistically indistinguishable, as we needed to show. This completes the proof. $\square$

D.2 The Corrupted Majority Case

If at least $\lceil n/2 \rceil$ parties are corrupted, then the simulator has to simulate an adversary that can decrypt all values that it sees. In this case, $\mathcal{S}_{\text{UVCDN}}^{\text{corrupt}}$ (ITM 3) performs a run of the actual protocol with respect to the adversary. It can do this, because it gets the inputs of the honest parties from $\mathcal{T}_{\text{VSFE}}$ (line 5); and it can compute their shares of the decryption key by Lagrange interpolation from the shares of the corrupted parties (line 5)[7]. Again, the simulation differs depending on whether or not $\mathcal{R}$ is corrupted.

The easiest case is if $\mathcal{R}$ and $\mathcal{V}$ are both corrupted (lines 33–36). In this case, no honest parties produce output, so we just need to ensure that the adversarial state at the end of the simulation is indistinguishable from a real execution. For this, we simply execute the real protocol with the given adversary, and return the resulting state. We also need to supply some values to the trusted party; but in this case, it is not important what values we send, because they do not influence the result of any honest party (lines 36).

If $\mathcal{R}$ is corrupted but $\mathcal{V}$ is not, then the simulator needs to ensure that the honest verifier that it simulates either outputs a real encryption of the result (for which we need to supply the randomness to $\mathcal{T}$), or $\perp$. As mentioned, the simulator can simply run the full protocol with respect to $\mathcal{A}$. Running the protocol with $\mathcal{A}$ is an algorithm, of which witness-extended emulation (Theorem 2) guarantees an emulator that returns witnesses for all provided proofs. The simulator runs this emulator (lines 25–29). The emulator extracts witnesses for all returned proofs; so in particular, unless it fails (which happens with negligible probability by Theorem 2), all proofs are for statements that are really in their respective languages. Moreover, we let the emulator return the inputs of all input parties, and the randomness for the encryption returned by vercomp; the simulator sends these values to $\mathcal{T}$ as appropriate.

If $\mathcal{R}$ is not corrupted, then the simulator runs the protocol with the adversary, until it reaches the output phase. Again, witness-extended emulation provides the simulator with the inputs of the corrupted parties; and it ensures that all proofs that the adversary provides are for statements that are in their respective languages. If the computation fails, the simulator sends $\perp$ to the trusted party (line 22). Now, for the output and proof phases, the simulator obtains encryption X of the output from the trusted party (line 13). If the computation does not fail, then it needs to make sure that the verifier outputs X. Analogously to the simulator in the honest case, $\mathcal{S}_{\text{UVCDN}}^{\text{corrupt}}$ chooses $D = (1 + N)^{-y} X$ so that, indeed,

[7] Actually, the simulator can only determine Δs_i because it cannot divide by Δ under unknown modulus $pp'qq'$; but one observes that the UVCDN protocol always uses s_i in combination with Δ, so knowing Δs_i is sufficient.

ITM 3 $\mathcal{S}^{\text{corrupt}}_{\text{UVCDN}}$: Simulator for Universally Verifiable CDN (corrupted majority)

```
// pre: $\mathsf{pk}$ public key; $C$ ($|C| \geq \lceil n/2 \rceil$) corrupted parties with secret keys
//          $\{s_i\}_{i\in\mathcal{P}\cap C}$, inputs $\{x_i\}_{i\in\mathcal{I}\cap C}$
// post: attacker simulated, values provided to $\mathcal{T}$ to mimic real execution outputs
$(r, \mathcal{O}) \leftarrow \mathcal{S}^{\text{corrupt}}_{\text{UVCDN},\mathcal{A}}(\mathsf{pk}{=}(N, v, v_0, \{v_i\}_{i\in\mathcal{P}}), C, \{x_i\}_{i\in\mathcal{I}\cap C}, \{s_i\}_{i\in\mathcal{P}\cap C}, a) :=$
    $\{x_i\}_{i\in\mathcal{I}\setminus C} := \mathsf{recv}(\mathcal{T}); \{\Delta s_i\}_{i\in\mathcal{P}\setminus C} :=$ ⟨Lagrange interpolation from $\{s_i\}_{i\in\mathcal{P}\cap C}$⟩
    if $\mathcal{R} \notin C$ then          // result party honest: receive output encryption from $\mathcal{T}$
        ⟨run UVCDN($\mathsf{pk}, \{s_i\}_{i\in\mathcal{P}}, \{x_i\}_{i\in\mathcal{I}}$) line 1–26 with adversary
          $\mathcal{A}(\mathsf{pk}, \{s_i\}_{i\in\mathcal{P}\cap C}, \{x_i\}_{i\in\mathcal{I}\cap C}, a)$ using witness-extended emulation; let $x'_i$
          be the extracted inputs by parties in $\mathcal{I} \setminus F$; let $\mathcal{A}$, $\mathcal{O}$ be the state of the
          adversary and random oracle at this point⟩
        forall $i \in \mathcal{I} \cap F$ do $x'_i := 0$
        $\mathsf{send}(\{x'_i\}_{i\in\mathcal{I}\cap C}, \mathcal{T})$
        $X := \mathsf{recv}(\mathcal{T})$
        if $|\mathcal{P} \setminus F| \geq \lceil n/2 \rceil$ then
            $y \in_R \mathbb{Z}_N; D := (1+N)^{-y}X; \mathcal{A}^{\mathcal{O}}(D)$
            $(\pi_{\text{PK}d}, \_) := \mathcal{S}_{\text{F}\Sigma}(\Sigma_{\text{PK}}, \{\mathcal{R}\}, \{D\})$
            $Y := X_{\text{outgate}} \cdot D^{-1}$
            foreach $i \in \mathcal{P} \setminus C$ do $Y_i := Y^{2\Delta s_i}$
            $\{Y_i\}_{i\in(\mathcal{P}\setminus F)\cap C} := \mathcal{A}^{\mathcal{O}}(\{Y_i\}_{i\in\mathcal{P}\setminus C})$
            $((\_, Y_0, \_, \_), \pi_{\text{CD}y}, \_) := \mathcal{S}_{\text{M}\Sigma}(\Sigma_{\text{CD}}, \Phi_{\text{CD}}, \mathcal{P} \setminus F, \{(Y, Y_i, v, v_i)\}_{i\in\mathcal{P}\setminus F}, D)$
            $\pi := (\{(D_c, E_c, \pi_{\text{CM}c}, S_{0,c}, \pi_{\text{CD}c})\}_{c\in\text{gates}}, (D, \pi_{\text{PK}d}, Y_0, \pi_{\text{CD}y}))$
        if $|\mathcal{P} \setminus F| < \lceil n/2 \rceil$ then $\mathsf{send}(\perp, \mathcal{T})$ else $\mathsf{send}(\top, \mathcal{T})$
        if $\mathcal{V} \in C$ then $\mathcal{A}^{\mathcal{O}}(\pi)$
    else if $\mathcal{V} \notin C$ then      // result party corrupted: check encryption verifier gets
        ⟨run UVCDN($\mathsf{pk}, \{s_i\}_{i\in\mathcal{P}}, \{x_i\}_{i\in\mathcal{I}}$) with adversary $\mathcal{A}(\mathsf{pk}, \{s_i\}_{i\in\mathcal{P}\cap C},$
         $\{x_i\}_{i\in\mathcal{I}\cap C}, a)$ using witness-extended emulation; let $x'_i$ be the extracted
         inputs by parties in $\mathcal{I} \setminus F$; let $s$ be $\perp$ if the verifier returns $\perp$, or otherwise
         the extracted randomness from line 17 of $\mathsf{vercomp}$; let $\mathcal{A}$, $\mathcal{O}$ be the state
         of the adversary and random oracle after the execution⟩
        forall $i \in \mathcal{I} \cap F$ do $x'_i := 0$
        $\mathsf{send}(\{x'_i\}_{i\in\mathcal{I}\cap C}, \mathcal{T}); \mathsf{recv}(\mathcal{T}); \mathsf{send}(s, \mathcal{T})$
    else    // result party and verifier corrupted: no honest party outputs anything
        ⟨run UVCDN($\mathsf{pk}, \{s_i\}_{i\in\mathcal{P}}, \{x_i\}_{i\in\mathcal{I}}$) with adversary
         $\mathcal{A}(\mathsf{pk}, \{s_i\}_{i\in\mathcal{P}\cap C}, \{x_i\}_{i\in\mathcal{I}\cap C}, a)$ giving $\mathcal{A}, \mathcal{O}$⟩
        forall $i \in \mathcal{I} \cap C$ do $x'_i := 0$
        $\mathsf{send}(\{x'_i\}_{i\in\mathcal{I}\cap C}, \mathcal{T}); \mathsf{recv}(\mathcal{T}); \mathsf{send}(\perp, \mathcal{T})$
    return $(\mathcal{A}^{\mathcal{O}}(), \mathcal{O})$   // return adv.-chosen output and simulated random oracle
```

the output $(1+N)^y D$ corresponds to X; and simulates the proofs of plaintext knowledge and correct decryption (lines 15-21).

We achieve statistical indistinguishability (computational indistinguishability would be sufficient):

Lemma 3. *For all inputs $x_1, \ldots, x_m, C, a$ with $|\mathcal{P} \cap C| > \lceil n/2 \rceil$, $\mathcal{S}^{corrupt}_{UVCDN,\mathcal{A}}$ is PPT, and*

$$EXEC_{UVCDN,\mathcal{A}}(k, (x_1, \ldots, x_m), C, a) \text{ and } IDEAL_{f,\mathcal{S}_{UVCDN,\mathcal{A}}}(k, (x_1, \ldots, x_m), C, a)$$

are statistically indistinguishable in security parameter k.

Proof. We remark that $\mathcal{S}^{\text{corrupt}}_{\text{UVCDN}}$ is clearly PPT since it only calls PPT subroutines.

Statistical indistinguishability is clear in case the result party is corrupted. Namely, the final adversarial state in the simulator is obtained by witness-extended emulation from the state of the adversary interacting with the true protocol. Hence, by Theorem 2, it is statistically indistinguishable from the adversarial state in the true protocol. Moreover, in both cases, a honest verifier outputs $(1+N)^r s^N$ for $D = (1+N)^* s^N$ from line 21 of vercomp, or $\perp$ if the computation failed.

Now, consider the case when the result party is honest. Then lines 1–26 of the protocol are executed in both the real-world and the ideal-world execution. Afterwards, instead of running the output phase of the actual protocol using a random encryption D, the simulator receives a random encryption X of the result from $\mathcal{T}$, and runs the output phase on $(1+N)^{-y}X$ for random y. The other difference is that proofs are simulated rather than executed. However, $(1+N)^{-y}X$ is identically distributed to D and simulation of proofs is statistically indistinguishable, so the adversarial state is statistically indistinguishable. Moreover, both in the real world and in the ideal world, the result party outputs (r, s) such that $X = (1+N)^r s^N$ and the verifier outputs X. Hence, the overall outputs are statistically indistinguishable. This concludes the proof. $\square$