VITĀRIT: Paying for Threshold Services on Bitcoin and Friends

Lucjan Hanzlik
CISPA Helmholtz Center for Information Security
hanzlik@cispa.de

Aniket Kate
Purdue University/Supra Research
aniket@purdue.edu

Easwar Vivek Mangipudi
Supra Research
e.mangipudi@supraoracles.com

Pratyay Mukherjee
Supra Research
pratyay85@gmail.com

Sri AravindaKrishnan Thyagarajan
University of Sydney
tsrikrishnan@gmail.com

***Abstract*—Blockchain service offerings have seen a rapid rise in recent times. Many of these services realize a decentralized architecture with a threshold adversary to avoid a single point of failure and to mitigate key escrow issues. While payments to such services are straightforward in systems supporting smart contracts, achieving fairness poses challenges in systems like Bitcoin, adhering to the UTXO model with limited scripting capabilities. This is especially challenging without smart contracts, as we wish to pay only the required threshold of $t+1$ out of the n servers offering the service together, without any server claiming the payment twice.**

In this paper, we introduce VITĀRIT [1], a novel payment solution tailored for threshold cryptographic services in UTXO systems like Bitcoin. Our approach guarantees robust provable security while facilitating practical deployment. We focus on the t-out-of-n distributed threshold verifiable random function (VRF) service with certain properties, such as threshold BLS signatures, a recently highlighted area of interest. Our protocol enables clients to request *verifiable random function (VRF)* values from the threshold service, triggering payments to up to $t+1$ servers of the distributed threshold VRF.

Our efficient design relies on simple transactions using signature verification scripts, making it immediately applicable in Bitcoin-like systems. We also introduce new tools and techniques at both the cryptographic and transaction layers, including a novel signature-VRF exchange protocol for standard constructions, which may be of independent interest. Additionally, our transaction flow design prevents malicious servers from claiming payments twice, offering broader implications for decentralized payment systems. Our prototype implementation shows that in the two-party interaction, the client takes 126.4 msec, and the server takes 204 msec, demonstrating practicality and deployability of the system.

1. Introduction

Today, clients frequently delegate their computational tasks to third-party services, driven by either the computational intensity of the tasks or the desire to maintain conceptual simplicity in the client's application. To mitigate the vulnerability of a single point of failure, these services often operate as distributed systems comprising a set of n servers, any $t < n$ of which can potentially be corrupted. The client's desired final output can be computed through a straightforward computation on the results obtained from any $t+1$ of the servers. This service configuration is referred to as having a t-out-of-n threshold structure, widely implemented in many real-world systems.

However, this service comes at a cost, requiring the client to provide monetary compensation. In the context of Web2, a trusted party, typically the service itself or an authority such as a bank or court, can be relied upon to ensure an atomic exchange between the computation output and the payment. Atomicity ensures that the service receives payment only if the client obtains the computation output successfully. However, depending on a trusted party for such exchanges is undesirable from a security standpoint. Therefore, there is strong advocacy for embracing Web3 or decentralized solutions, which eliminate the need for reliance on a central authority, enhancing the overall security of the exchange.

Blockchain-based solutions have prominently emerged as the focal point for facilitating client-service exchanges [1], where users leverage smart contracts on blockchain systems such as Ethereum. For a better understanding, let us consider the following two representative applications.

Application 1: Randomness Service. In an online event, such as a game, a client may require random coins (referred to as randomness) for active participant engagement, with applications extending to enhancing the security of critical infrastructure in the Web3 ecosystem [2]. To obtain the desired randomness, the client typically enlists the services of a third party. The associated smart contract facilitates an atomic transfer of d coins (belonging to the client) to the service, contingent upon receiving a *valid* randomness within a time frame. While the service could employ various cryptographic mechanisms [3], [4], [5], [6] for randomness generation, the widely adopted choice for public verification, as required by the contract, is *verifiable random functions*

1. A Sanskrit word for 'distributed'

(VRF) [7], [8], a cryptographic primitive to generate randomness and publicly verifiable proof. If the service operates on a t-out-of-n threshold structure, it employs *distributed VRF (DVRF)* [9], [8], [10] where each server generates partial values, and $t+1$ of them collectively contribute to generating the final randomness and the proof.

Application 2: Oracle Service. In the second scenario, the client seeks a digital attestation for a real-world event denoted by m from a service referred to as an *oracle*. Depending on the specific nature of m, this oracle may take the form of a notary, insurer, trading house, or another authoritative entity entrusted with attesting to real-world occurrences. In the Web3 ecosystem [11], [12], [10], these oracle services have become integral, serving as gateways for accessing real-world information to be incorporated into blockchain smart contracts. To compensate for the service, the client initiates a smart contract with the message m hard-coded, and the payment of x coins is only executed upon receiving a *valid* attestation from the oracle. In a threshold setting, the client has the flexibility to seek attestations from any $t+1$-out-of-n independent oracles or want a single attestation computed when $t+1$ oracles provide their partial values. Both scenarios can be supported for an atomic payment-attestation exchange through appropriate verification steps embedded in the smart contract.

Limitations of Smart Contracts. Unfortunately, the smart contract-based exchange comes with multiple issues:

- *Privacy*: Smart contracts deployed on public blockchains expose sensitive information about clients, third-party services, requests, and associated applications to the entire network, raising privacy concerns. This transparency allows malicious players to launch MeV attacks [13], and potentially censor transactions, posing a threat to exchange atomicity and making the system vulnerable to DoS attacks. The community has extensively explored these challenges, offering insights and solutions applicable to various blockchain applications [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25].
- *Efficiency*: The smart contract-based approach involves blockchain nodes performing complex cryptographic operations to verify the "validity" of the service's response. However, running such operations on smart contracts is significantly more expensive than just regular transaction verification, both in terms of verification time and monetary costs like gas costs (in Ethereum) or transaction fees. For example, to verify a BLS-based VRF [9], one needs a pairing verification check, which is well known to be computationally expensive, costing 113K gas on Ethereum.
- *Compatibility*: Smart contract-based exchange solutions face compatibility challenges with prominent blockchain systems like Bitcoin, Stellar [26], Ripple [27], Monero [28], Zcash [29], etc. Opting for solutions that minimize contract or script usage proves advantageous in achieving broader compatibility with these major blockchain systems. Moreover, such an approach also contributes to enhanced privacy and cost-effectiveness, as discussed earlier.

Bitcoin Compatibility: A One-Shot Solution. Bitcoin, a pioneer in the Web3 ecosystem [30], has garnered enduring credibility. Recently, there is a surge in interest to develop Bitcoin-compatible protocols for diverse blockchain applications [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [31]. New scalability solutions like Bitcoin Rollups, and BitVM[2] have expanded the ambit of Bitcoin, prompting further exploration of Bitcoin's potential. Developers and researchers are leveraging Bitcoin's robust infrastructure to address smart contract limitations by minimizing contracts and prioritizing privacy. These protocols expose only essential information on the blockchain, using standard transactions with minimal scripts like signature verification. The core principle is that secure, efficient protocols built on Bitcoin's simple scripts can be widely applied across blockchain systems while preserving privacy. Therefore, we ask the question:

Can we build a privacy-preserving Bitcoin-compatible protocol that enables atomic payments in exchange for computational tasks provided by a threshold service, like a randomness service?

In this work, we answer this question by presenting an efficient Bitcoin-compatible protocol for randomness services whose template can also be applied to other services. We propose several new techniques to overcome challenges arising from the threshold nature of the service and the atomicity requirement in the ensuing payment. We now summarize the contributions made in this work.

1.1. Our contribution

Firstly, we introduce the first formal security model for the threshold service-payment atomic exchange between a client and a t-out-of-n threshold service. Here, the threshold service is computing some keyed function f on the client's input request. Assuming *any* $t+1$ servers are honest and available,[3] and the adversary can corrupt a maximum of t servers including the client, our model outlines the conditions for final output evaluation. Atomicity requires that only *the first* $t+1$ (correctly) responding servers are eligible for payment. We rely on the *Universal Composability (UC)* framework [32] to establish a robust foundation for this atomic exchange. Next, we present VITĀRIT, the first Bitcoin-compatible, i.e., without Turing complete scripts, atomic exchange solution for a distributed verifiable threshold service (DVTS). VITĀRIT uses standard transactions and signature verification scripts within Bitcoin, representing the most minimal script/contract implementation across blockchain systems. The modular design of our protocol incorporates various novel cryptographic building blocks and ideas in blockchain payment design. At its core, VITĀRIT makes novel use of a new encryption scheme called *verifiable non-committing encryption (VNE)*. VNE is an encryption scheme

2. a Bitcoin Virtual Machine to run and verify complex proofs

3. In most settings, the service requires an honest majority, i.e., $t+1 > \frac{n}{2}$, but our model only requires $t+1$ honest servers where t is the corruption threshold.

where one can verify if the message encrypted satisfies certain useful properties. On a high level, VNE has a *non-committing property* that helps in the security analysis to show that an adversarial client learns an honest server's partial evaluation value *if and only if* they behaved honestly during the interaction, and learn no information otherwise.

Next, we instantiate VITĀRIT for the case of a t-out-of-n distributed VRF (DVRF) service (where $t < \frac{n}{2}$) as the underlying DVTS. For this, we primarily instantiate our VNE with an efficient construction involving novel cut-and-choose proof techniques complemented by optimizations via batching, which may be of independent interest. Compared to a SNARK proof-based construction, which may have efficient communication, our instantiation avoids inefficient and theoretically unsound computation similar to [25], where a random oracle is treated as an arithmetic circuit.

Finally, we evaluate our VITĀRIT protocol using a rust prototype implementation (see Section 9). The client interacts with each of the servers individually; the two-party interaction takes 126.4 msec for the client and 204.67 msec for the server, showing the practicality and efficiency of the protocol. The server posts one transaction on the blockchain to claim the funds; the client obtains the partial VRF value using the data from the posted transaction. To demonstrate that our protocol improves even protocols using smart contracts, we realize the threshold VRF functionality on Ethereum through the traditional smart contract approach and one using VITĀRIT. Even after ignoring the high deployment cost of the traditional approach, the usage of VITĀRIT results in consuming approximately 290K gas for five servers, which is a more than 68% cost saving compared to the prior approach consuming 932K gas. This improvement only increases with increasing number of servers.

2. Related Work

The exploration of using blockchains to achieve fairness in multi-party computations originated with Bentov and Kumaresan [33]. They demonstrated the utilization of smart contracts on blockchains to ensure fair information exchange among parties. Simultaneously, within the blockchain ecosystem, there emerged the concept of cryptocurrency-based contingent payments. This involves users exchanging a digital secret for payment in a way that ensures the seller gets paid only if the secret is revealed to the buyer within a specific timeframe. The smart contract facilitating this exchange is known as a *hash timelock contract (HTLC)* [34], [35]. These contingent payments have found several applications in blockchain systems over the years, notably the Lightning Network [36]. All of these works laid the foundation for a blockchain-based marketplace where clients and services can interact with the guarantee of atomicity.

Recently, a successful line of academic and industry works has delved into the topic of smart contract minimization. Malavolta et al. [14] present a cryptographic machinsm called *anonymous multihop locks* for realizing payment channel networks using on signature verification scripts from the blockchain. In a follow-up work, Thyagarajan and Malavolta [15] expand the cryptographic toolset to help realize payment channel networks across all blockchain systems without using any smart contracts. Subsequently, many Bitcoin-compatible scalability improvement solutions [18], [16] have been proposed that ensure similar levels of privacy. Coin mixing, which is a method to anonymize coins in a blockchain-based cryptocurrency has also been studied in the Bitcoin-compatible setting [22], [23], [25] with efficient and minimal script solutions. Thyagarajan, Malavolta, and Moreno-Sanchez [21] proposed *universal atomic swaps* that allowed users to atomically exchange their coins without relying on any smart contracts. Madathil et al. [24] proposed a solution for oracle-based conditional payments in blockchains. The above applications are some of the many examples where we only knew of solutions that relied on smart contracts. All of these works introduce new cryptographic techniques to abstract away the smart contract logic and let only simple transactions and signatures go on the blockchain.

Accountable threshold signatures [37], [38] is another cryptographic primitive where the final threshold signature can be traced to the servers that contributed signature shares for its reconstruction. We can hope to extend the functionality to any threshold service and make payments to the traced servers. However, the primitive as such requires a trusted combiner party to combine the threshold shares and is not directly compatible with the threshold services currently in use. But we foresee using accountable threshold services as an interesting future direction.

3. Solution Overview

In this section, we present an overview of the challenges, and techniques involved in the design of our model, and the construction of an efficient Bitcoin-compatible protocol. We will start with a brief look at the system model.

3.1. System Model and Setup

We consider a system with a client party A and a t-out of-n threshold randomness service with n servers, denoted by $S_1, \ldots, S_n$. The network is fully connected and synchronous; any message forwarded by an honest party reaches all honest parties within a time limit of δ. An adversary can corrupt the client A and up to t servers of the service at the beginning of the protocol and the corruption is static. All the non-corrupted parties are honest, continuously available, and have secure authenticated communication channels with each other. The client and the servers also have access to a public blockchain, where they can read and write transactions.

Distributed Verifiable Threshold Service - Random Function Service. For ease of presentation in this section, we will use the Distributed Verifiable Random Function (DVRF) service as the candidate threshold service DVTS. A verifiable random function [7] can be seen as the public-key version of a keyed cryptographic hash function $F_{sk}(\cdot)$, where a trusted party evaluates $F_{sk}(m)$ on inputs m in such a way that the output v can publicly be verified via a proof π_v using a verification algorithm $\mathsf{Verify}()$. The final output and the proof are unique for a given m, and the output itself is pseudorandom. A distributed (or threshold) verifiable random function [9] fundamentally distributes the trust among n parties. Instead of one party holding the secret key sk, n parties or servers hold shares of the secret key sk or sk^{DVRF} to make the context of DVRF explicit. The servers execute a (possibly distributed) key generation to generate the joint verification key vk^{DVRF} (corresponding to sk^{DVRF}), where the i-th server obtains the share $(sk_i^{\mathsf{DVRF}}, vk_i^{\mathsf{DVRF}})$ of the key pair. When evaluating the VRF on input m, the i-th server employs $\mathsf{PartEval}()$ using its secret key share sk_i^{DVRF}, resulting in a partial evaluation v_i^{DVRF} and a proof π_i^{DVRF}. One can verify if the partial evaluations are correct using $\mathsf{PartVerify}()$. Combining $t+1$ partial evaluations using $\mathsf{Combine}()$ algorithm produces the final VRF value v^{DVRF} and a proof π^{DVRF}. With the verification key vk^{DVRF}, input m, output v^{DVRF}, and proof π^{DVRF}, anyone can publicly verify the correctness of the VRF computation using the $\mathsf{Verify}()$ algorithm. The security guarantee ensures pseudorandomness even if up to t servers are corrupted [9].

We are now ready to describe the system setup. The service verification key vk^{DVRF} and the server's partial verification keys $(vk_1^{\mathsf{DVRF}}, \ldots, vk_n^{\mathsf{DVRF}})$ are publicly available, including the client A. The servers keep the secret key shares privately; the secret key sk^{DVRF} is only known collectively by at least $t+1$ servers. The client has input $m = m^*$, and wants $F_{sk}(m^*)$ from the service along with the proof π_{m^*}. For convenience we assume that the proof is part of the output in this exposition.

3.2. Atomic Payment-Randomness Exchange

At the core, the client seeks payment assurance only when the final VRF output $F_{sk}(m^*)$ is disclosed. Conversely, the service intends to unveil the final VRF output solely upon receiving payment. However, dealing with payment in a threshold service, structured as t-out-of-n, introduces complexity: Without a designated leader, each of the n servers holds an equal position. Consequently, payments must be directly transferred to these servers, posing a nuanced challenge.

Attempt 1: Pay all servers. We can have a mechanism where the client's payment is distributed to all servers equally. However, unless $t = n-1$, and if in fact $t+1 < n$, we have a "freeriding" problem where a set of up to $n-t-1$ freeriding servers might get paid without contributing any "DVRF work". While being unfair and hurting the incentive structure, this approach also makes the mechanism highly resource-inefficient by overburdening the $t+1$ working servers.

Attempt 2: Pay only the working servers. In this scenario, the exchange protocol will evenly distribute the payment among the fastest responding $t+1$ servers. Let's break down this protocol into three phases for clarity: (1) deposit, (2) exchange, and (3) payout.

Deposit phase: The client initiates a deposit of $d \cdot (t+1)$ coins through transaction tx_{Setup} into address addr_D. This address is jointly controlled by the client and the service (as a whole) to prevent unilateral fund withdrawal. To support the threshold structure of the service, addr_D is set to be controlled by *any* server. More concretely, the address consists of two public keys, pk_A (client) and pk_S (service), with the client holding the secret key for pk_A and *each* server holding the secret key for pk_S. Any transaction spending from addr_D must be signed with pk_A and pk_S. Moreover, refunding from addr_D to the client is possible after a predefined timeout $\mathbf{T}$.

Exchange phase: Post deposit, the client requests all the servers (whichever ones respond) to perform a VRF on m^*. Each server locally runs its own DVRF partial evaluation on $m*$ and broadcasts the partial results to the other servers. With $t+1$ honest servers online, any server can combine partial values to obtain the final VRF output. For simplicity, let us assume that servers $S_1, \ldots, S_{t+1}$ were the fastest responding $t+1$ servers. Now, any server (which need not be from the first $t+1$ servers), acting as a *combiner* that combined the values, can contact the client claiming to hold the final VRF value. The client and the combiner set up a payment transaction tx_{pay} for coin distribution from addr_D to servers $S_1, \ldots, S_{t+1}$. The combiner can sign the transaction as it holds the secret key sk_S (corresponding to pk_S). To finalize the payment, the combiner has to obtain the client's signature on tx_{pay}. To fairly exchange the final VRF output and the signature of the client on tx_{pay}, the client and the combiner can execute a special purpose 2-party protocol Γ^*, which we will revisit in more detail later in the section.

Remark 1. Note that the system has at least $t+1$ available servers; the client does not check which servers are alive. The first $t+1$ servers pick themselves by the order in which they publish the payment transaction on the blockchain, spending the $t+1$ deposits.

Remark 2. Throughout this paper, all interaction between the client and each server is via private communication channels.

Payout phase: The combiner, having learned the client's signature, can publish the transaction tx_{pay} on the blockchain along with the two signatures required to spend. By the fairness of Γ^*, the client obtains the VRF output. While the above solution looks natural, it suffers from the following issues.

- *Malicious combiner:* The first security vulnerability is when the combiner is malicious. In this scenario, the corrupt combiner can manipulate payment transactions (tx'_{pay}) to exclusively redeem payments for itself or choose any $t+1$ servers. This vulnerability is twofold. Firstly, any server can potentially become a combiner, including a malicious one. Secondly, the malicious combiner can exploit a loophole—avoiding any "DVRF work" by not generating partial evaluations—and still gain access to the final VRF value through $t+1$ honest servers. The combiner can now choose its choice of $t+1$ servers and negotiate a payment for them. This, however, deprives all honest servers of any payment, constituting a form of a "denial of payment" attack.

- *Malicious servers:* An attacker corrupting a subset of servers can create conflicting views to honest servers regarding the list of fastest responding servers. To resolve these divergent views, one must rely on a trusted combiner to report the correct list of servers or a distributed consensus mechanism to decide on the correct list. This necessitates an extra trust assumption or involves more intricate and complex mechanisms with higher communication and computation complexity.

Therefore, we explore an alternative approach where the client bypasses the combiner and pays the working servers directly.

Attempt 3: Pay the working servers directly. Here, the client directly interacts with all the servers and compensates only the fastest responding $t+1$ servers. The exchange protocol follows the same three phases as the previous attempt.

Deposit phase. The client deposits a total of $d \cdot (t+1)$ in the form of $t+1$ independent deposits, each containing x coins. Similar to the previous attempt, joint control is established between the client and all servers for each deposit, while the client can also refund unspent deposits after a timeout $\mathbf{T}$.

Remark 3. Note that at this point, the client does not know which are the fastest $t+1$ servers, and therefore, the $t+1$ deposits are independent of this information and are only dependent on the common service public key pk_S. If, at any point, a server does not respond, the client simply continues its interaction with the other servers.

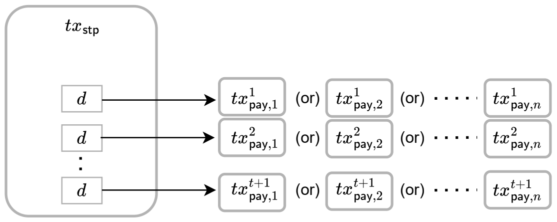
tx_{stp}
d
d
d
$tx^1_{\mathsf{pay},1}$ (or) $tx^1_{\mathsf{pay},2}$ (or) ···· $tx^1_{\mathsf{pay},n}$
$tx^2_{\mathsf{pay},1}$ (or) $tx^2_{\mathsf{pay},2}$ (or) ···· $tx^2_{\mathsf{pay},n}$
$tx^{t+1}_{\mathsf{pay},1}$ (or) $tx^{t+1}_{\mathsf{pay},2}$ (or) ···· $tx^{t+1}_{\mathsf{pay},n}$

Figure 1. Transaction flow sketch. Server S_j has $t+1$ possible payment transactions that it can use to get paid.

Exchange phase. Upon initiating the VRF request with the message m^*, the client negotiates payment transactions individually with each server. An essential consideration is that the i-th deposit is not exclusively assigned to the i-th server; any server S_j (where $i \neq j$) has the potential to redeem the i-th deposit. Consequently, for $i \in \{1, \ldots, t+1\}$ and $j \in \{1, \ldots, n\}$, the client formulates the payment transaction $tx^i_{\mathsf{pay},j}$, transferring d coins from the i-th deposit to the address of server S_j (see Figure 1). Each server S_j, independently computes the DVRF partial evaluation on m^*. Subsequently, it interacts with the client to exchange the partial output for the client's signature on $tx^i_{\mathsf{pay},j}$ for $i \in \{1, \ldots, t+1\}$. This involves executing a special purpose 2-party exchange sub-protocol Γ^*, akin to the previous attempt, with the exception that now it is for server S_j's DVRF partial value v_j^{DVRF} and the client's signature on the payment transaction $tx^i_{\mathsf{pay},j}$.

Payout phase. Server S_j publishes transaction $tx^{i^*}_{\mathsf{pay},j}$ for some $i^* \in \{1, \ldots, t+1\}$ on the blockchain, accompanied by its signature on the transaction and the client's signature obtained from the sub-protocol. As a reminder, server S_j can sign the transaction since it holds joint control of the i^*-th deposit. Like the previous attempt's payout phase, the client locally obtains the partial DVRF value v_j^{DVRF} (from the 2-party protocol Γ^*) and t other partial DVRF values from t other servers. With $t+1$ partial values, the client can learn the final VRF output v^{DVRF}.

While the protocol guarantees payment for the working servers, a closer examination reveals the possibility of overpayment for a working server. It's important to note that a server cannot know in advance which i^*-th deposit it will successfully redeem. Consequently, to maximize its chances of payment, a server will try to redeem all $t+1$ deposits. There is a high probability that the server will succeed in multiple attempts and receive payments more than once. This unintended consequence may prevent honest servers from receiving payments. If we had smart contracts, we could easily solve this issue by letting the contract keep track of the keys of the paid servers and disallowing double payments. However, keeping track of paid servers when all servers can potentially redeem all deposits is tricky in our setup.

VITĀRIT - Pay the working servers exactly once directly. We introduce a new auxiliary address-based payment mechanism to address the challenge of multiple successful redemption attempts and build our solution VITĀRIT. This approach ensures that servers can try to redeem all $t+1$ deposits but succeed in at most one attempt.

Here's a simple idea: Each server S_i locks a small amount of coins ($\epsilon = 1$ Satoshi) in an auxiliary address $\mathsf{addr}_{\mathsf{aux},i}$, which includes the server's public key $pk_{\mathsf{aux},i}$. Each payment transaction $tx^j_{\mathsf{pay},i}$ for $j \in 1, \ldots, t+1$ is now *bound* to spending not only from the j-th deposit address but also simultaneously from the server's auxiliary address $\mathsf{addr}_{\mathsf{aux},i}$ (as illustrated in Figure 2). In an honest scenario, to publish $tx^j_{\mathsf{pay},i}$, server S_i only needs to post the signature w.r.t. $pk_{\mathsf{aux},i}$ since it knows the corresponding secret key.

Due to the *unspent transaction output (UTXO)* model of Bitcoin, if $\mathsf{addr}_{\mathsf{aux},i}$ is spent using $tx^j_{\mathsf{pay},i}$, then no other payment transaction $tx^k_{\mathsf{pay},i}$ for $j \neq k$ can be published on the blockchain. This ensures that a server can get paid at most once. While the server may have multiple such auxiliary addresses, the UTXO model ensures that the transaction $tx^j_{\mathsf{pay},i}$ is bound to exactly one specific $\mathsf{addr}_{\mathsf{aux},i}$ (created by $tx_{\mathsf{aux},i}$ as shown in Figure 2) of the server S_i.

3.3. 2-Party Protocol Γ^*: Partial VRF-Signature Exchange

A key element in VITĀRIT involves a specialized 2-party protocol Γ^* where the client and the server exchange the

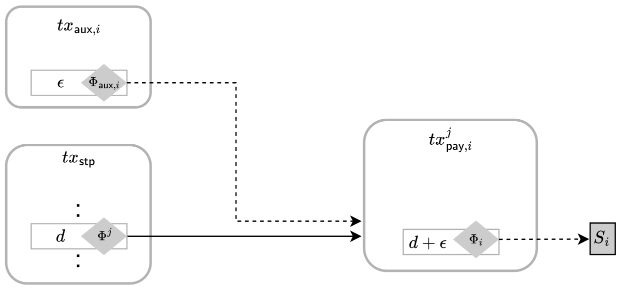

Figure 2. Transaction flow for the payment. We assume it is the i-th server that gets paid from the j-th deposit. The payment transaction $tx^j_{\mathsf{pay},i}$ spends from the outputs of transactions $tx_{\mathsf{aux},i}$ and tx_{Setup} simultaneously.

client's signature on the payment transaction and the server's partial VRF value.

This protocol is constructed using two cryptographic tools: adaptor signatures [39] and a novel tool introduced in this work called *verifiable non-committing encryption (VNE)*. At a high level, adaptor signatures allow users to pre-sign a message m using the signing key sk and a statement Y from the NP relation R, generating a pre-signature $\hat{\sigma}$. This pre-signature is not yet a valid signature but can be adapted into one using the witness y corresponding to the statement Y. The witness y can also be efficiently extracted from the pre-signature $\hat{\sigma}$ and the adapted signature σ.

To better grasp the intuition behind VNE, consider a standard public key encryption scheme and sketch the 2-party protocol.

1) The server S_i generates an encryption-decryption key pair (ek, dk), and sends a ciphertext ct to the client that encrypts the partial VRF value v_i^{DVRF}.
2) The client uses the encryption key ek as the statement Y of the adaptor NP relation such that the corresponding decryption key dk is the witness y. The client sends a pre-signature $\hat{\sigma}$ on the payment transaction $tx^j_{\mathsf{pay},i}$, $j \in [t+1]$, generated w.r.t. Y, and the client's signing key.
3) The server uses the witness $y = dk$ to adapt $\hat{\sigma}$ and obtain the client's signature σ and later publish it on the blockchain along with $tx^j_{\mathsf{pay},i}$ to get the payment.
4) The client reads the blockchain and locally extracts the witness $y = dk$ using the adaptor signature extraction.
5) The client can decrypt ct using dk and learn the partial VRF value v_i^{DVRF} of S_i.

Verifying the messages transmitted by the respective recipients is crucial for the security of the above exchange. More specifically, the client verifies the correct encryption of the partial VRF value in the ciphertext ct, while the server ensures the proper generation of the pre-signature $\hat{\sigma}$ through efficient verification enabled by adaptor signatures. To achieve the former, the encryption scheme must allow efficient ciphertext verification, which the VNE achieves: The encryption scheme is defined w.r.t. an NP relation R where we encrypt a witness wit of an instance $inst$, such that $(inst, wit) \in R$. The relation R in our case is the relation between the partial VRF values and DVRF verification keys. The resulting ciphertext ct can be verified efficiently to check if it correctly encrypts a witness of the instance $inst$. VNE also guarantees a non-committing property which is critical for the security analysis as discussed below.

Non-Committing property of VNE. When proving security against a corrupt client, we want to ensure that the adversary cannot learn any information about the partial value v_i^{DVRF} if a valid signature is unavailable to the server. The verifiability property of the adaptor signature ensures that if the pre-signature sent by the client is verified successfully, then the honest server is guaranteed to receive the correct signature (by extracting using its partial DVRF evaluation). Therefore, we need to care about the case when the adversary does not send a valid pre-signature. Here, the only information the client has about v_i^{DVRF} is in the ciphertext ct. A straightforward idea is to replace ct with a fake random ciphertext ct' independent of v_i^{DVRF} in step (1). We can hope to rely on the security of the encryption scheme to do this step in an indistinguishable manner. Additionally, the fake ciphertext ct' must 'verify' successfully to the adversary, for which we can rely on standard zero-knowledge-style simulation arguments.

However, analyzing the adversary's behavior during the ciphertext switch is challenging. If the ciphertext is switched to a fake one, the adversary can distinguish the views, but later, the adversary submits a valid pre-signature in step (2), and subsequent steps proceed honestly. In the context of VITĀRIT, we also cannot guess efficiently if the adversary succeeds in step (2) or not, because there are an exponential number of possible combinations of $n-t$ servers where the adversary can fail in step (2). Hence, we refine the encryption scheme to be non-committing: A simulator can simulate the ciphertext ct' in step (1) without prior knowledge of v_i^{DVRF}. If the adversary succeeds in step (2), the simulator explains the ciphertext ct' in a way that, after step (5), the client decrypts the correct v_i^{DVRF}, mirroring an honest run. The non-committing property ensures indistinguishability between the simulated and real views.

Importantly, if the adversary fails in step (2), the simulator isn't obligated to explain, relying on the encryption scheme's hiding property. Ultimately, we guarantee that an adversary without generating a valid pre-signature in step (2) cannot extract any information about v_i^{DVRF}.

Constructing VNE. Let us now see how to construct such a VNE efficiently. To encrypt a witness $wit = v_i^{\mathsf{DVRF}}$, the encryption algorithm samples $r, s \leftarrow \mathbb{Z}_p$ and returns a ciphertext $ct := (c, \pi)$ where $c := (c_1, c_2, c_3)$, such that $c_1 := g^r, c_2 := ek^r \cdot g^s, c_3 := H(g^s) \oplus wit$. Here $ek = g^{dk}$ is the encryption key, H is a hash function and π is a non-interactive zero-knowledge proof (NIZK) for the relation:

$$R := \left\{ \begin{array}{l} \left(\left((c_1, c_2, c_3), vk^{\mathsf{DVRF}}, \left(vk_\ell^{\mathsf{DVRF}}\right)_{\ell \in [n]}, m^*\right), (v_i^{\mathsf{DVRF}}, r, s)\right) : \\ s.t.\ ,\ (v_i^{\mathsf{DVRF}}, \cdot) \leftarrow \mathsf{DVRF.PartEval}(sk_i^{\mathsf{DVRF}}, m^*) \wedge \\ c_1 := g^r, c_2 := ek^r \cdot g^s, c_3 := H(g^s) \oplus v_i^{\mathsf{DVRF}} \end{array} \right\}$$

where $\left(vk^{\mathsf{DVRF}}, \left(vk_\ell^{\mathsf{DVRF}}, sk_\ell^{\mathsf{DVRF}}\right)_{\ell \in [n]}\right)$ is generated honestly by running $\mathsf{DVRF.DKgen}(1^\lambda, t, n)$. The proof certifies that c is well-formed with respect to the correct partial VRF value corresponding to the i-th verification key vk_i^{DVRF}. We can verify the ciphertext ct by verifying the proof π and to decrypt given the decryption key dk, one can simply return $wit := c_3 \oplus H(c_2 \cdot (c_1^{dk})^{-1})$. If we model H as a programmable random oracle in the security analysis, we can show that the above construction is non-committing.

Efficient Construction. One of the main disadvantages of the above construction is that we use a circuit representation of the random oracle H. This technique is acceptable and used in practice, i.e., proving knowledge of the pre-image of the hash function-based random oracle using SNARKs [40], [41]. Unfortunately, the implications of this approach on formal security have not been well studied. Therefore, we will discuss an efficient construction that circumvents this issue using the well-known cut-and-choose techniques.

More concretely, the VNE ciphertext will now consist of group elements and non-committing encryption ciphertexts generated in the following way. As part of the VNE ciphertext, we add values $A_j := g^{a_j}$, $B_j := g^{b_j}$, and ciphertext $ct_j := (c_{1,j}, c_{2,j}, c_{3,j})$ generated as above (in Section 3.3) that encrypts the value $A_j^{b_j}$ (instead of wit). Here we have $j \in [2\lambda_s]$ for statistical security parameter λ_s.

The cut-and-choose technique proceeds as follows: We compute the challenge set J by hashing all tuples $H_c\left((A_j, B_j, ct_j)_{j\in[2\lambda_s]}\right)$ (akin to Fiat-Shamir transform). The set $[2\lambda_s]/J$ are the indices we "open" to the verifier. In other words, we prove they were computed according to the protocol. We provide a_j, b_j, and the randomness r_j used in the ciphertext ct_j for all $j \in [2\lambda_s]/J$. For the unopened set $j \in J$, we add additional values $Z_j := A_j^{b_j} \cdot wit$, and a NIZK proof π_j that Z_j is well-formed, where $wit := v_i^{\mathsf{DVRF}}$. The proof π_j can be easily constructed using simple pairing checks. Putting it all together, the final VNE ciphertext ct consists of all the values (A_j, B_j, ct_j), along with the opened and unopened values.

To validate the VNE ciphertext, we first check if all the opened values are correct using simple canonical checks, and for the unopened values we check the NIZK proofs π_j. If all checks are successful, the cut-and-choose guarantees that with overwhelming probability, for some $j \in [2\lambda_s]/J$, the ciphertext ct_j is well-formed and can be correctly decrypted.

To decrypt the ciphertext ct, the decryption algorithm picks the $j \in [2\lambda_s]/J$, such that ct_j is correctly formed. Using the decryption key dk, it decrypts ct_j to reveal $A_j^{b_j}$. It finally outputs the witness wit, by using $A_j^{b_j}$ to unblind the value Z_j.

Curve compatibility and batching. A key advantage of our VNE construction is that the key pair (ek, dk) for the inner encryption scheme can be based on the same group $\mathbb{G}_p$, as supported by the adaptor signature scheme (e.g., a Schnorr group). In contrast, the witness wit and the values Z_j for $j \in [2\lambda_s]/J$ can be based on the group $\mathbb{G}_q$ that supports pairings. We also present additional batching techniques for our VNE construction in the full version Section 8.2.1, where the client and the server interact to exchange witnesses for N instances rather than just one. The idea is reminiscent of the batching technique introduced by Lindell and Riva [42] and explored by Madathil et al. [24] and brings down the communication cost from scaling linearly in λ_s to scaling linearly in a smaller constant $1 + \frac{\lambda_s}{1+\log N}$.

We discuss further interesting extensions of VITĀRIT in Section 8.3

4. Preliminaries

We denote by $\lambda \in \mathbb{N}$ the security parameter and by $x \leftarrow \mathcal{A}(\mathsf{in}; r)$ the output of the algorithm $\mathcal{A}$ on input in using $r \leftarrow\!\!\$ \{0,1\}^*$ as its randomness. We often omit this randomness and only mention it explicitly when required. We say that an algorithm is (non-uniform) PPT if it runs in probabilistic polynomial time. We say that a function is *negligible* if it vanishes faster than any polynomial. We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$ (for $n \in \mathbb{N}$). A tuple of values is denoted by the vector notation $\mathbf{v} = (v_1, v_2 \ldots)$. The notation $\overset{?}{=}$ denotes an equality check and if the check is unsuccessful, the respective algorithm aborts the execution and reports failure.

For payments, transactions are generated by the transaction function tx. A transaction tx_A, denoted

$$tx_A := tx\begin{pmatrix}[(\mathsf{addr}_1, \Phi_1, v_1), \ldots, (\mathsf{addr}_n, \Phi_n, v_n)],\\ [(\mathsf{addr}'_1, \Phi'_1, v'_1), \ldots, (\mathsf{addr}'_m, \Phi'_m, v'_m)]\end{pmatrix},$$

charges v_i coins from each input address addr_i for $i \in [n]$, and pays v'_i coins to each output address addr'_j where $j \in [m]$. Here Φ_i and Φ'_j are scripts that encode the conditions to spend the coin from the associated addresses. **Primer on the UTXO Transaction Model.** In an *Unspent Transaction Output* (UTXO) model, the coins are stored in addresses denoted by $\mathsf{addr} \in \{0,1\}^\lambda$ and addresses are spendable (i.e., used as input to a transaction) *exactly once*. Transactions can be posted on the blockchain to transfer coins from a set of input addresses to a set of output addresses (excluding transaction fees). More precisely, transactions are generated by the transaction function tx. A transaction tx_A, denoted

$$tx_A := tx\begin{pmatrix}[(\mathsf{addr}_1, \Phi_1, v_1), \ldots, (\mathsf{addr}_n, \Phi_n, v_n)],\\ [(\mathsf{addr}'_1, \Phi'_1, v'_1), \ldots, (\mathsf{addr}'_m, \Phi'_m, v'_m)]\end{pmatrix},$$

charges v_i coins from each input address addr_i for $i \in [n]$, and pays v'_i coins to each output address addr'_j where $j \in [m]$. It must be guaranteed that $\sum_{i\in[n]} v_i \geq \sum_{j\in[m]} v'_j$. The difference $f = \sum_{i\in[n]} v_i - \sum_{j\in[m]} v'_j$ is offered as the transaction fee to the miner who includes the transaction in his block.

An address is typically associated with a *script* $\Phi : \{0,1\}^\lambda \to \{0,1\}$ which states what conditions need to be satisfied for the coins to be spent from the address. A transaction is considered authorized if it is attached with witnesses $[x_1, \ldots, x_n]$ such that $\Phi_i(x_i) = 1$ (publicly computable) for all $i \in [n]$. In this work, we only require the scripts Φ to be signature verification algorithms with the public key hard-coded in them.

Public-Key Encryption Scheme. A public-key encryption scheme $\mathsf{PKE} := (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ allows one to generate a key pair $(ek, dk) \leftarrow \mathsf{KGen}(1^\lambda)$ that allows anyone to encrypt messages as $c \leftarrow \mathsf{Enc}(ek, m)$ and allows only the owner of the decryption key dk to decrypt ciphertexts as $m \leftarrow \mathsf{Dec}(dk, c)$. We require perfect correctness and standard CPA security from PKE. We additionally want a non-committing property for the encryption scheme, that intuitively says that there exists a simulator Sim that can create a pair of an encryption key ek and ciphertext c, indistinguishable from a real pair encryption key/ciphertext. Given any message m at any later point in time, the simulator can output the randomness r that explains the transcript (ek, c) for m, i.e., $c := \mathsf{Enc}(ek, m; r)$. For formal definitions, refer [43].

Digital Signatures. A digital signature scheme DS, has a key generation algorithm $\mathsf{KGen}(1^\lambda)$ that takes the security parameter 1^λ and outputs the public/secret key pair (pk, sk), a signing algorithm $\mathsf{Sign}(sk, m)$ that inputs sk and a message $m \in \{0,1\}^*$ and outputs a signature σ, and a verification algorithm $\mathsf{Vf}(pk, m, \sigma)$ that outputs 1 if σ is a valid signature on m under pk, and outputs 0 otherwise. We require strong unforgeability from the signature scheme which is the standard notion of security

Adaptor Signatures. Adaptor signatures [39] let users generate a pre-signature on a message m that can be adapted into a valid signature using some secret value. For this, the primitive is defined w.r.t. a hard relation R.

Definition 1 (Adaptor Signatures). An adaptor signature scheme AS w.r.t. a hard relation R and a signature scheme $\mathsf{DS} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$ has algorithms $(\mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVf}, \mathsf{Ext})$ defined as:

- $\hat{\sigma} \leftarrow \mathsf{pSign}(sk, m, Y)$: the pre-sign algorithm takes as input a signing key sk, message $m \in \{0,1\}^*$ and statement $Y \in \mathcal{L}_R$, outputs a pre-signature $\hat{\sigma}$.
- $0/1 \leftarrow \mathsf{pVf}(vk, m, Y, \hat{\sigma})$: the pre-verify algorithm takes as input a verification key vk, message $m \in \{0,1\}^*$, statement $Y \in \mathcal{L}_R$ and pre-signature $\hat{\sigma}$, outputs either 1 (for valid) or 0 (for invalid).
- $\sigma/\perp \leftarrow \mathsf{Adapt}(\hat{\sigma}, y)$: the adapt algorithm takes as input a pre-signature $\hat{\sigma}$ and witness y, outputs a signature σ or $\perp$.
- $y \leftarrow \mathsf{Ext}(\sigma, \hat{\sigma}, Y)$: the extract algorithm takes as input a signature σ, pre-signature $\hat{\sigma}$ and statement $Y \in \mathcal{L}_R$, outputs a witness y such that $(Y, y) \in R$, or $\perp$.

An adaptor signature scheme AS has pre-sign algorithm $\mathsf{pSign}(sk, m, Y)$ that takes as input a a signing key sk, message m and statement $Y \in \mathcal{L}_R$, outputs a pre-signature $\hat{\sigma}$. The pre-verify algorithm $\mathsf{pVf}(vk, m, Y, \hat{\sigma})$ takes as input a verification key vk, message m, statement Y and pre-signature $\hat{\sigma}$, outputs either 1 (for valid) or 0 (for invalid). The adapt algorithm $\mathsf{Adapt}(\hat{\sigma}, y)$ takes as input a pre-signature $\hat{\sigma}$ and witness y, outputs a signature σ. The extract algorithm $\mathsf{Ext}(\sigma, \hat{\sigma}, Y)$ takes as input a signature σ, pre-signature $\hat{\sigma}$ and statement Y, outputs a witness y such that $(Y, y) \in R$.

For security, we want unforgeability even when the adversary is given access to pre-signatures w.r.t. the signing key sk. We also require that, given a pre-signature and a witness for the instance, one can always adapt the pre-signature into a valid signature (*pre-signature adaptability*). Finally, we require that, given a valid pre-signature and a signature with respect to the same instance, one can efficiently extract the corresponding witness (*witness extractability*). For formal definitions, we refer the reader to [44].

Hard Relations. We denote by $\mathcal{L}_R$ the associated language defined as $\mathcal{L}_R := \{Y \mid \exists y,\ (Y, y) \in R\}$. The relation is called a hard relation if the following holds: (i) There exists a PPT sampling algorithm $\mathsf{GenR}(1^\lambda)$ that outputs a statement/witness pair $(Y, y) \in R$; (ii) For all PPT adversaries $\mathcal{A}$ the probability of $\mathcal{A}$ on input Y outputting a witness y is negligible. In this work, we use the discrete log language $\mathcal{L}_{\mathsf{DL}}$ defined with respect to a group $\mathbb{G}$ with generator g and order p. The language is defined as $\mathcal{L}_{\mathsf{DL}} := \{Y \mid \exists y \in \mathbb{Z}_p,\ Y = g^y\}$ with corresponding hard relation R_{DL}.

Non-Interactive Zero Knowledge Proofs. Let $R : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ be a n NP-witness-relation with corresponding NP-language $\mathcal{L} := \{stmt \mid \exists wit \text{ s.t. } R(stmt, wit) = 1\}$. A non-interactive zero-knowledge proof (NIZK) [45] system for the relation R is initialized with a setup algorithm $\mathsf{Setup}(1^\lambda)$ that, on input the security parameter, outputs a common reference string crs and a trapdoor td. A prover can show the validity of a statement x with a witness w by invoking $\mathsf{Prove}(crs, x, w)$, which outputs a proof π. The proof π can be efficiently checked by the verification algorithm $\mathsf{Vf}(crs, x, \pi)$. We require a NIZK system to be (1) *zero-knowledge*, where the verifier does not learn more than the validity of the statement x i.e., there exists a simulator algorithm $\mathsf{Sim}(crs, \mathsf{td}, x)$ that without any information of the witness, w can convince the verifier to output 1, and (2) *sound*, where it is hard for any prover to convince a verifier of an invalid statement (chosen by the prover).

5. Formal Model

All the guarantees of our approach are captured by the ideal functionality $\mathcal{F}_{\mathtt{swap}}$, detailed in Figure 3. The ideal functionality interacts with client $\mathcal{C}$, the set of servers $\{S_j\}_{j \in [n]}$, and the simulator $\mathcal{S}_{\mathtt{swap}}$. It keeps track of the following variables $\{\mathcal{J}, R, \mathcal{I}, \mathcal{B}\}$, all initialized to $\perp$ (or $\emptyset$) implicitly. It also maintains different lists $Keys, SKeys, Inp, Out, Paid$ as required. The functionality has access to a key function $\mathsf{f}(\cdot, \cdot)$ with various interfaces for key generation, key sharing, evaluating the function, aggregating partial evaluations, and verifying the function evaluation. This function represents the threshold function being evaluated by the servers in the real world. The functionality controls a public key address pk_{sid} and transfers funds to that address when needed.

Adversary's capabilities. The adversary can corrupt the client and up to t servers, with at least $t + 1$ servers being honest and always available. Corrupted parties interact with the functionality via the simulator $\mathcal{S}_{\mathtt{swap}}$, which notifies the functionality of client corruption via a `corrupt-client` message.

During key generation, the functionality $\mathcal{F}_{\mathtt{swap}}$ gets the list of corrupted servers C from the simulator. It aborts if

Ideal functionality $\mathcal{F}_{\texttt{swap}}$

- The functionality $\mathcal{F}_{\texttt{swap}}$ interacts with the client $\mathcal{C}$, the server $\mathcal{I} := \{S_j\}_{j\in[n]}$, and the simulator $\mathcal{S}_{\texttt{swap}}$. $\mathcal{F}_{\texttt{swap}}$ has access to the functionality which realizes a keyed function $\mathsf{f}(\cdot,\cdot)$. $\mathsf{f}()$ has interfaces `keygen, keyshare, eval, aggregate, verify` to generate key pair, share the secret key, evaluate the function, aggregate multiple evaluations and verify each evaluation of the function.
  - $\mathcal{I}$ is the set of all servers offering the service. $\mathcal{J}$ is the set of all server indices the client wants to initiate the swap with.$\mathcal{B}$ is the set of all server indices with which the client wishes to abort the interaction (initialized to ϕ). R is the list of all server indices responding with partial output.
  - $\mathit{Keys}, \mathit{SKeys}, \mathit{Inp}, \mathit{Out}, \mathit{Paid}$ are initialized to empty lists.

Client Corruption Upon receiving $(\texttt{corrupt-client}, C)$ from $\mathcal{S}_{\texttt{swap}}$, mark the client $\mathcal{C}$ as `corrupt`.

Key Generation Upon receiving $(\texttt{keygen}, \mathsf{sid}, \mathrm{C})$ from $\mathcal{S}_{\texttt{swap}}$, do the following:

1) Define $\mathrm{C}_{\mathcal{I}} := \mathrm{C} \cap \mathcal{I}$ and $\mathrm{H}_{\mathcal{I}} := \mathcal{I} \setminus \mathrm{C}_{\mathcal{I}}$ and set $n_{\mathcal{I}} := |\mathcal{I}|$
   a) If $n_{\mathcal{I}} < 2t+1$, exit. Else, continue
   b) If $|\mathrm{C}_{\mathcal{I}}| > t$, mark $\mathcal{I}$ as $\mathsf{corrupt}$ and exit. Else, continue
2) Sample a key pair $(sk, vk) \leftarrow \mathsf{f}.\texttt{keygen}(1^{\lambda})$
3) Generate shares of the generated key pair $\{(sk_i, vk_i)\}_{i\in\mathcal{I}} \leftarrow \mathsf{f}.\texttt{keyshare}(sk, vk)$
4) Append (sk, vk) to $\mathit{Keys}[\mathcal{I}, \mathsf{sid}]$. For each $S_i \in \mathcal{I}$, append (sk_i, vk_i) to $\mathit{SKeys}[S_i]$
5) Send $(\texttt{keygen}, \mathsf{sid}, vk, \mathcal{I}, \{(sk_i, vk_i)\}_{i\in\mathrm{C}_{\mathcal{I}}})$ to $\mathcal{S}_{\texttt{swap}}$and $(\texttt{keygen}, \mathsf{sid}, vk, \mathcal{I})$ to each $S_i \in \mathrm{H}_{\mathcal{I}}$

Client deposit and input. After receiving $(\texttt{swap}_c, \mathsf{sid}, vk, m, (t+1)d, sk_d, pk_d, \mathcal{J})$ from $\mathcal{C}$, or from $\mathcal{S}_{\texttt{swap}}$ in case $\mathcal{C}$ is marked as `corrupt`, do the following:

1) If $\mathit{Inp}[vk, m] = \perp$, and $(\cdot, vk) \in \mathit{Keys}[\cdot, \mathsf{sid}]$, then set $\mathit{Inp}[vk, m] := \mathcal{C}$ and forward the message $(\texttt{swap}_c, \mathsf{sid}, vk, m, (t+1)d, pk_d, \mathcal{J})$ to $\mathcal{S}_{\texttt{swap}}$. Else, exit
2) If $\mathcal{S}_{\texttt{swap}}$ returns the same message, do the following. Else, exit
   a) Call the subroutine $\texttt{freeze}(sid, (t+1)d, sk_d, pk_d, pk_{sid})$. If the `freeze` call is unsuccessful, abort. Else, continue
   b) Send $(\texttt{swap}_c, \mathsf{sid}, vk, m, d)$ to each $S_i \in \mathrm{H}_{\mathcal{I}}$

Client selective abort. Upon receiving $(\texttt{abort}, \mathsf{sid}, m, j)$ from $\mathcal{C}$ at any point in time, update $\mathcal{B} \leftarrow \mathcal{B} \cup \{j\}$ and call $\texttt{unfreeze}(\mathsf{sid}, pk_d, sk_d, d)$.

Server partial evaluation. • Upon receiving the message $(\texttt{swap}_s, sid, vk, m, j, vk_j, rk_j, y_j)$ from $\mathcal{S}_{\texttt{swap}}$, if $j \in \mathcal{J}, j \in \mathrm{C}_{\mathcal{I}}$ and $(\cdot, vk_j) \in \mathit{SKeys}[S_j]$, do the following:

   1) Run $\phi \leftarrow \mathsf{f}.\texttt{verify}(y_j, vk_j)$
   2) If $\phi = 1$, do the following. Else, do nothing.
      a) Append j to the list R
      b) Set $\mathit{Out}[vk_j, m] := y_j$
      c) If $j \notin \mathcal{B}$, $j \notin \mathit{Paid}[\mathcal{I}, \mathsf{sid}]$ and $|\mathit{Paid}[\mathcal{I}, \mathsf{sid}]| \leq t+1$, then call $\texttt{transfer\_pay}(\mathsf{sid}, rk_j, d)$ and set $\mathit{Paid}[\mathcal{I}, \mathsf{sid}] := \mathit{Paid}[\mathcal{I}, \mathsf{sid}] \cup \{j\}$

- Upon receiving the message $(\texttt{swap}_s, sid, vk, m, j, vk_j, rk_j)$ from S_j, if $j \in \mathcal{J}$ and $j \in \mathrm{H}_{\mathcal{I}}$, do the following:
   1) Send (sid, j) to $\mathcal{S}_{\texttt{swap}}$
   2) Compute $y_j \leftarrow \mathsf{f}.\texttt{eval}(sk_j, m)$, set $\mathit{Out}[vk_j, m] := y_j$
   3) If $j \notin \mathcal{B}$, $j \notin \mathit{Paid}[\mathcal{I}, \mathsf{sid}]$ and $|\mathit{Paid}[\mathcal{I}, \mathsf{sid}]| \leq t+1$, then call $\texttt{transfer\_pay}(sid, rk_j, d)$ and set $\mathit{Paid}[\mathcal{I}, \mathsf{sid}] := \mathit{Paid}[\mathcal{I}, \mathsf{sid}] \cup \{j\}$

Client Obtaining VRF values Upon receiving $\{\texttt{obtain\_vrf}, sid, m\}$ from $\mathcal{C}$ or $\mathcal{S}_{\texttt{swap}}$,

- If $\mathcal{C}$ is `corrupt`, and the message is received from $\mathcal{S}_{\texttt{swap}}$, do the following:
  - For each $j \in \mathit{Paid}[\mathcal{I}, sid]$, append (j, y_j) to the set O where $\mathit{Out}[vk_j, m] = (y_j)$
  - Forward (y, vk, O) to $\mathcal{S}_{\texttt{swap}}$.
- Else
   1) Run $y := \mathsf{f}.\texttt{eval}(sk, m)$, append y to $\mathit{Eval}[vk, m]$ and Forward (y, vk) to $\mathcal{C}$.

Verification: Upon $(\texttt{verify}, vk, m, y, \pi)$ from $\mathcal{C}$,

1) If there is an $\mathcal{I}$ for which $(\cdot, vk) \in \mathit{Keys}[\mathcal{I}]$ *and* $\mathit{Eval}[vk, m]$ is defined then do the following:
   a) If $(y, \pi) \in \mathit{Eval}[vk, m]$, set $f := \mathsf{f}.\texttt{verify}(y, \pi, vk)$. Else, set $f := 0$.
2) Else, set $f := 0$.
3) Finally return f to $\mathcal{C}$.

The subroutines are described here:

- $\texttt{freeze}(sid, v, pk_d, sk_d, pk_{sid})$: Transfers value v from pk_d to pk_{sid} using sk_d via $\texttt{Post}(sid, pk_d, pk_{sid}, v)$. The function is successful if the transaction is accepted.
- $\texttt{unfreeze}(sid, pk, pk_{sid}, v)$ - Transfers the frozen value held by pk_{sid} back to pk, via $\texttt{Post}(sid, pk_{sid}, pk, v)$.
- $\texttt{transfer\_pay}(sid, rk_k, d)$ - Transfers the value d to the server receiving key rk_k from pk_{sid}.

Figure 3: Ideal functionality of the `swap` protocol - the client pays only $t+1$ servers

the number of corrupted parties is a majority. Otherwise, it generates a secret and public key pair (sk, vk). The functionality also generates shares sk_i of the secret key sk and the corresponding public keys $vk_i, i \in [n]$. The generated key pairs (sk_i, vk_i) are stored and associated with each server $SKeys$. The functionality forwards the secret key share and corresponding verification keys of the corrupted parties to the simulator and verification keys to each non-corrupted party.

The client $\mathcal{C}$ deposits $(t+1) \cdot d$ coins to the public key pk_d and forwards both the secret key sk_d and the public key pk_d to the functionality. It also forwards the input mand the server set $\mathcal{J}$ with which the client wants to interact. Forwarding the secret key sk_d ensures the client can not withdraw the funds prematurely before paying the servers. They also indicate the verification key vk against which the requested output should be verified. Once the swap request is forwarded to and acknowledged by the simulator, the functionality transfers the deposit to a public key pk_{sid} and freezes it. If the freeze is unsuccessful, the protocol is aborted. If not, a swap request signal is forwarded to all the non-corrupt parties.

At any point before payment, the client can abort interaction with a server by sending an `abort` signal. The functionality tracks aborted servers, unfreezing corresponding deposit values using `unfreeze` for each abort.

The simulator forwards the server partial evaluations y_j on behalf of the corrupted parties through the swap_S message. The functionality $\mathcal{F}_{\texttt{swap}}$ verifies the values and stores them to the set of (partial) outputs. It also receives the evaluation signals from the honest parties and stores them against their verification keys vk_j and input x. For the honest parties, the functionality computes the partial evaluation and stores it. The functionality checks if each of the servers from which a value or signal is obtained is in the abort list. If no, and if the server is not paid and if a total $t+1$ servers are paid till that time, the server forwards the payment to the server using `transfer_pay` to the receiving address rk_j specified by the server or the simulator using a `Post(·)` call.

A non-corrupt client or the simulator can forward the obtain_vrf for the input x. If the client is corrupt, the functionality collects all the partial values of the servers that have been paid and forwards them to the simulator. Otherwise, it computes the final function output for the secret key sk and forwards it to the client. The functionality also allows the client to forward the `verify` message to request the functionality to verify the received final output.

The functionality immediately pays the servers which forward the values to ensure that the servers are paid exactly in the order in which they respond with the values. It also ensures that a maximum of $t+1$ servers are paid. The client can not obtain partial values and abstain from paying the server fee. Also, for aborted interactions, the deposit is returned to the client. The functionality forwards all the partial evaluations to the corrupt client; the final value is directly computed and forwarded to an honest client.

5.1. Distributed Verifiable Threshold Service.

We generalize the notion of threshold primitives, that are verifiable, such as threshold VRF, threshold signatures, etc. Here we have n parties who generate keys, possibly in a distributed fashion, such that any $t+1$ parties only collectively know the key. They perform partial evaluations on an input and given any $t+1$ of such partial values, one can combine them to obtain the final value. Additionally, all the partial and final values are publicly verifiable. For simplicity, we consider non-interactive interfaces except for the key generation phase, which can be distributed using specific protocols (e.g. [46]).

Definition 2. A (t, n)-distributed verifiable threshold service DVTS consists of five PPT algorithms $(\mathsf{DKgen}, \mathsf{PartEval}, \mathsf{PartVerify}, \mathsf{Combine}, \mathsf{Verify})$ that are defined below:

- $(vk, (vk_j, sk_j)_{j \in [n]}) \leftarrow \mathsf{DKgen}(1^\lambda, t, n)$: the distributed key generation algorithm is a (possibly interactive) algorithm executed among n parties. It takes as input the security parameter 1^λ, parameters t and n, and returns a global verification key vk to all and partial verification-secret key pairs (vk_j, sk_j) to the j-th party.
- $(v_i, \pi_i) \leftarrow \mathsf{PartEval}(sk_i, m)$: the partial evaluation algorithm takes as input a secret key share sk_i, and a message m, and returns partial evaluation v_i, and a proof of partial evaluation π_i.
- $0/1 \leftarrow \mathsf{PartVerify}(i, vk, (vk_j)_{j \in [n]}, m, v_i, \pi_i)$: the partial verification algorithm takes as input an index i, the global verification key vk, the partial verification keys $(vk_j)_{j \in [n]}$, a message m, the partial evaluation v_i and the proof of partial evaluation π_i, and returns 1 indicating valid, or returns 0 indicating invalid.
- $(v, \pi) \leftarrow \mathsf{Combine}(vk, (vk_j)_{j \in [n]}, m, \{(k_i, v_{k_i}, \pi_{k_i})_{i \in [t+1]}\})$: the combine algorithm takes as input the global verification key vk, the partial verification keys $(vk_j)_{j \in [n]}$, a message m, and a set of $t+1$ tuples where the i-th tuple consists of an index k_i, a partial evaluation v_{k_i}, and a proof of partial evaluation π_{k_i}. It returns an evaluation result v and a proof π.
- $0/1 \leftarrow \mathsf{Verify}(vk, (vk_j)_{j \in [n]}, m, v, \pi)$: the full verification algorithm takes as input the global verification key vk, the partial verification keys $(vk_j)_{j \in [n]}$, a message m, the evaluation v and the proof of evaluation π, and returns 1 indicating valid, or returns 0 indicating invalid.

We require $(t, n) -$ DVTS to satisfy correctness that guarantees for honestly generated keys and partial evaluations, any $t+1$ of them can be combined to generate a valid full evaluation value. We additionally want *unpredictability*, which intuitively says that an adversary with the knowledge of t honestly generated keys cannot predict the full evaluation on a message m^* for which it has not obtained any honest partial evaluation. Finally, DVTS must satisfy *robustness* that says if $t+1$ partial evaluations given by an adversary verify successfully, then the combined full evaluation must also verify successfully. The formal definitions of the properties are described in Appendix 6

$\underline{\mathsf{ExpPriv}^{\mathcal{A}}_{\mathsf{DVTS},t+1,n}(\lambda)}$:
$C \leftarrow \mathcal{A}(\lambda)$
$(vk, (vk_j, sk_j)_{j\in[n]}) \leftarrow \mathsf{DKgen}(1^\lambda, t, n)$
$(m^*, v, \pi) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{PartEval}}}(vk, (vk_j, sk_j)_{j\in C})$
$\mathcal{O}_{\mathsf{PartEval}}(S, m)$ returns $(v_i, \pi_i) \leftarrow \mathsf{PartEval}(sk_i, m)$ for all $i \in S \subseteq [n] \setminus C$
Winning condition: Output 1 if $\mathsf{Verify}(vk, (vk_j)_{j\in[n]}, m^*, v, \pi) = 1$ and m^* was not queried to $\mathcal{O}_{\mathsf{PartEval}}$.

Figure 4: Unpredictability experiment for DVTS.

6. Properties of DVTS

Definition 3 (Correctness). A (t, n)-distributed verifiable threshold service DVTS is said to be correct if for all $\lambda \in \mathbb{N}$, all $(vk, (vk_j, sk_j)_{j\in[n]}) \leftarrow \mathsf{DKgen}(1^\lambda, t, n)$, all messages $m \in \{0, 1\}^\lambda$, all $(v_i, \pi_i) \leftarrow \mathsf{PartEval}(sk_i, m)$ for any $i \in [n]$, it holds that:

- For all $i \in [n]$ $\Pr\big[\mathsf{PartVerify}(i, vk, (vk_j)_{j\in[n]}, m, v_i, \pi_i) = 1\big] = 1$
- For any $K \subseteq [n]$, where $|K| = t + 1$, we have $\Pr\big[\mathsf{Verify}(vk, (vk_j)_{j\in[n]}, m, v, \pi) = 1\big] = 1$ where $(v, \pi) \leftarrow \mathsf{Combine}(vk, (vk_j)_{j\in[n]}, m, \{(k, v_k, \pi_k)_{k\in K}\})$.

Definition 4 (Unpredictability). A (t, n)-distributed verifiable threshold service DVTS is said to be unpredictable if there exists a negligible function negl, for all $\lambda \in \mathbb{N}$, all PPT adversaries $\mathcal{A}$, the following holds:

$$\Pr\Big[\mathsf{ExpPriv}^{\mathcal{A}}_{\mathsf{DVTS},t+1,n}(\lambda) = 1\Big] \le \mathsf{negl}(\lambda)$$

where ExpPriv is described in Figure 4.

Definition 5 (Robustness). A $((t), n)$-distributed verifiable threshold service DVTS is said to be robust if for all $\lambda \in \mathbb{N}$, for all $(vk, (vk_j, sk_j)_{j\in[n]}) \leftarrow \mathsf{DKgen}(1^\lambda, t, n)$, there exists a negligible function negl, such that for any PPT adversary $\mathcal{A}$ that returns $m, (k_1, v_{k_1}, \pi_{k_1}), \ldots, (k_{t+1}, v_{k_{t+1}}, \pi_{k_{t+1}})$ and the following holds simultaneously with probability at most negl.

- For all $k \in \{k_1, \ldots, k_{t+1}\} \subset [n]$,
$$\mathsf{PartVerify}(k, vk, (vk_j)_{j\in[n]}, m, v_k, \pi_k) = 1$$
- $\mathsf{Verify}(vk, (vk_j)_{j\in[n]}, m, v, \pi) = 0$, where
$$(v, \pi) \leftarrow \mathsf{Combine}(vk, (vk_j)_{j\in[n]}, m, \{(k_i, v_{k_i}, \pi_{k_i})_{i\in[t+1]}\})$$

Instantiations. We remark that the above generic primitive captures many threshold / distributed primitives satisfying a structural requirement[4] similar to BLS signatures [47].[5] Among them the most prominent ones are the BLS-based distributed VRFs [9], [10], [8]. Additionally, this structural similarity extends to other primitives like threshold Boneh-Franklin identity-based key-derivation [48], as evidenced by a recent proposal [49] for a verifiable scheme; variants of threshold signatures [50], [51], [52] certain distributed PRFs [53] etc. We notice that, at the core of all these primitives, two things are common: (i) the communication pattern is one request from a client and one response from each server, two rounds in total over a star network, where servers don't communicate; (ii) the public verifiability of the individual and final (after combining) evaluations. It is worth noting that, DVTS does not require a compact final output – for example, a Combine procedure can just be an identity function. We present VITĀRIT w.r.t. this generic primitive, to ensure that any such instantiations would be immediately compatible.

4. As mentioned earlier, we want a 'non-interactive' reconstruction of the final output given partial evaluations with all evaluations - partial and full - to be publicly verifiable. These properties are supported by the BLS signature-based cryptographic objects (but not, e.g., Threshold Schnorr).

5. For BLS threshold or multi-signatures, it is easy to see that the properties are achieved straightforwardly.

7. VITĀRIT Protocol

In this section, we formally describe our VITĀRIT protocol. Recall that we have a client party A and a $(t+1)$-out of-n threshold service. We make use of the following cryptographic tools:

1) A signature scheme $\mathsf{DS} := (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$ that is used to sign transactions on the blockchain.
2) A $(t+1)$-out-of-n or $(t+1, n)$-(Non-Interactive) Verifiable Distributed Threshold Service scheme $\mathsf{DVTS} := (\mathsf{DKgen}, \mathsf{PartEval}, \mathsf{PartVerify}, \mathsf{Combine}, \mathsf{Verify})$.
3) An adaptor signature scheme $\mathsf{AS} := (\mathsf{KGen}, \mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVf}, \mathsf{Ext})$ that is defined with respect to the signature scheme DS and discrete logarithm hard relation $\mathcal{R}_{\mathsf{DL}}$.

We use an additional tool called *verifiable non-committing encryption* scheme that we introduce below.

7.1. Verifiable Non-Committing Encryption

We consider a efficiently sampleable NP relation R with the sampling algorithm $\mathsf{GenR}(1^\lambda)$ that returns an instance $inst$, a witness wit, and some private auxiliary information z, such that $(inst, wit) \in R$. We now define a verifiable non-committing encryption scheme with respect to the relation R. The definition is inspired by the work of Brakerski et al. [43].

Definition 6 (Verifiable Non-Committing Encryption). A verifiable non-committing encryption scheme VNE is defined with respect to a NP relation R, and consists of algorithms $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{VfEnc}, \mathsf{Dec})$ defined as:

- $(ek, dk) \leftarrow \mathsf{KGen}(1^\lambda)$: the key generation algorithm that inputs the security parameter λ and outputs an encryption key ek and a decryption key dk.
- $ct \leftarrow \mathsf{Enc}(ek, inst, wit, z)$: the encryption algorithm takes as input the encryption key ek, an instance $inst$, a witness wit, and an auxiliary information z. It returns a ciphertext ct.

- $0/1 \leftarrow \mathsf{VfEnc}(ek, inst, ct)$: the verification algorithm takes as input the encryption key ek, an instance $inst$, and a ciphertext ct. It returns 1 for valid, otherwise returns 0.
- $wit \leftarrow \mathsf{Dec}(dk, inst, ct)$: the decryption algorithm takes as input a decryption key dk, an instance $inst$, and a ciphertext ct. It returns a witness wit.

Remark 4. The GenR algorithm outputs values $inst$, wit and z, and they are generated from the protocol in which VNE interfaces are executed as a sub-routine (Figure 8, steps [4-6]).

We provide the definitions of correctness, simulatability and soundness of VNE in Section B

7.1.1. Constructing VNE. To construct our VNE scheme, we make use of a public-key encryption scheme PKE := (KGen, Enc, Dec) described in Figure 5. The encryption scheme is similar to the hashed Elgamal which works with a group $\mathbb{G}_p$ of prime order p, and a generator g_p. It additionally uses a hash function $H_m : \mathbb{G}_p \to \{0,1\}^{|m|}$ where $|m|$ is the size of messages being encrypted. It is folklore [54] that the encryption scheme is non-committing when H_m is modeled as a random oracle and the simulator in the simulatability definition can program H_m. Henceforth we denote the encryption scheme as $\mathsf{PKE}^{\mathsf{nc}}$ to distinguish the scheme from standard PKE schemes.

$\underline{\mathsf{KGen}(1^\lambda)\text{: The key generation algorithm does the following:}}$
- Sample $x \leftarrow \mathbb{Z}_p$
- Set $ek := g_p^x$, $dk := x$
- Return (ek, dk)

$\underline{\mathsf{Enc}(ek, m)}$: The encryption algorithm does the following:
- Sample $r, s \leftarrow \mathbb{Z}_p$
- Set $c_1 := g_p^r, c_2 := (ek)^r \cdot g_p^s, c_3 := H_m(g_p^s) \oplus m$
- Return $c := (c_1, c_2, c_3)$

$\underline{\mathsf{Dec}(dk, c)}$: The decryption algorithm does the following:
- Parse $c := (c_1, c_2, c_3)$
- Return $m' := c_3 \oplus H_m\left(c_2/c_1^{dk}\right)$

Figure 5: The concrete non-committing $\mathsf{PKE}^{\mathsf{nc}}$ scheme

We additionally need a NIZK proof [45], [55] for the NP language $\mathcal{L}$ described below:

$$\mathcal{L} := \left\{ \begin{matrix} (c, ek, inst) : \ \exists\ (wit, r)\ s.t.\ , \\ c \leftarrow \mathsf{PKE}^{\mathsf{nc}}.\mathsf{Enc}(ek, wit; r) \ \wedge (inst, wit) \in R \end{matrix} \right\}$$

With the above $\mathsf{PKE}^{\mathsf{nc}}$ scheme and the NIZK proof, we give a generic construction of VNE for any NP relation R in Figure 6. Note that the Enc algorithm ignores the auxiliary information z entirely. The auxiliary information plays a more vital role in the concrete instantiation which we discuss in Section 8. The below theorem states the security of the VNE construction (Figure 6). The proof is deferred to Section C.

Theorem 1. Let $\mathsf{PKE}^{\mathsf{nc}}$ be a secure non-committing public key encryption scheme. Let $(\mathsf{Setup}_{\mathcal{L}}, \mathsf{Prove}_{\mathcal{L}}, \mathsf{Vf}_{\mathcal{L}})$ be a NIZK proof system for the language $\mathcal{L}$. The construction from Figure 6 is a secure VNE scheme with respect to the relation R.

We make use of the $\mathsf{PKE}^{\mathsf{nc}}$ (Figure 5) and NIZK for language $\mathcal{L}$.

$\underline{\mathsf{KGen}(1^\lambda)\text{: The key generation algorithm does the following:}}$
- Sample $(ek, dk) \leftarrow \mathsf{PKE}^{\mathsf{nc}}.\mathsf{KGen}(1^\lambda)$
- Return (ek, dk)

$\underline{\mathsf{Enc}(ek, inst, wit, z)}$: The encryption algorithm does the following:
- Sample r from randomness space of $\mathsf{PKE}^{\mathsf{nc}}$ (Figure 5)
- Compute $c \leftarrow \mathsf{PKE}^{\mathsf{nc}}.\mathsf{Enc}(ek, wit; r)$
- Compute $\pi \leftarrow \mathsf{Prove}_{\mathcal{L}}(crs_{\mathcal{L}}, (c, ek, inst), (wit, r))$
- Return $ct := (c, \pi)$

$\underline{\mathsf{VfEnc}(ek, inst, ct)}$: The verification algorithm does the following:
- Parse $ct := (c, \pi)$
- Return $\mathsf{Vf}(crs_{\mathcal{L}}, (c, ek, inst), \pi)$

$\underline{\mathsf{Dec}(dk, inst, ct)}$: The decryption algorithm does the following:
- Parse $ct := (c, \pi)$
- Return $wit \leftarrow \mathsf{PKE}^{\mathsf{nc}}.\mathsf{Dec}(dk, c)$

Figure 6: Construction of VNE scheme

7.2. VITĀRIT Protocol Flow

The payment protocol is described in Figure 7. As outlined in Section 3, we have a setup phase where party A first creates t addresses and deposits x coins into each address. Let the j-th address be denoted by public keys $(pk_{A,j}, pk_{S,j})$ and script ϕ_j that requires the spending transaction to be signed with respect to both the public keys $pk_{A,j}$ (of party A) and $pk_{S,j}$ (of the servers). Note that every server knows the secret key $sk_{S,j}$ that is designated to spend the j-th deposit. Server S_i also deposits ϵ coins into an auxiliary address with the public key $pk_{\mathsf{aux},i}$ and it knows the corresponding secret key $sk_{\mathsf{aux},i}$. The script $\phi_{\mathsf{aux},i}$ associated with the auxiliary address is a straightforward script that requires the spending transaction to be signed with respect to the public key $pk_{\mathsf{aux},i}$.

In the payment phase, when party A and server S_i interact, they generate a payment transaction for each of the t deposits. Let us consider the j-th deposit where $j \in [t+1]$. Each of these transactions also simultaneously spends from the auxiliary address $pk_{\mathsf{aux},i}$ of server S_i. Party A chooses a request message m^* and engages with server S_i in a two-party sub-protocol $\tilde{\Gamma}^{\mathsf{2PC}}_{\mathsf{DVTS,DS}}$ to exchange a payment from A and server S_i's partial evaluation value. More specifically, the server S_i learns the signature of party A from $\tilde{\Gamma}^{\mathsf{2PC}}_{\mathsf{DVTS,DS}}$ and publishes the payment transaction spending the j-th deposit, along with its own signatures (for $pk_{S,j}$ and $pk_{\mathsf{aux},i}$). In the next step, party A uses the information published on the blockchain, specifically, the signature of party A that was published by the server, to extract the partial value $(i, v_i^{\mathsf{DVTS}}, \pi_i^{\mathsf{DVTS}})$.

After successful interactions with $t+1$ servers, where the servers published the payment transactions and got paid from all of the party A's deposits, party A has $t+1$ partial values. Without loss of generality, let us assume the first $t+1$ servers were successful in the above two-party sub-protocol. In this case, party A has extracted $((1, v_1^{\mathsf{DVTS}}, \pi_1^{\mathsf{DVTS}}), \ldots, (t+1, v_{t+1}^{\mathsf{DVTS}}, \pi_{t+1}^{\mathsf{DVTS}}))$, and it can combine the values using $\mathsf{DVTS.Combine}()$ to obtain the final value v^{DVTS}.

7.2.1. Two-party Sub-protocol $\Gamma^{\mathsf{2PC}}_{\mathsf{DVTS,DS}}$. The two-party protocol $\Gamma^{\mathsf{2PC}}_{\mathsf{DVTS,DS}}$ (described in Figure 8) is executed between server S_i and party A to exchange the partial value and party A's signature on the payment transaction. We rely on the VNE scheme for the relation R_i for $i \in [n]$, which is defined as,

$$R_i := \left\{ \begin{matrix} \left(\left(vk^{\mathsf{DVTS}}, \left(vk_j^{\mathsf{DVTS}}\right)_{j\in[n]}, m^*\right), v_i^{\mathsf{DVTS}}\right) : \ s.t.\ , \\ v_i^{\mathsf{DVTS}} \leftarrow \mathsf{DVTS.PartEval}(sk_i^{\mathsf{DVTS}}, m^*) \end{matrix} \right\}$$

where $\left(vk^{\mathsf{DVTS}}, \left(vk_j^{\mathsf{DVTS}}, sk_j^{\mathsf{DVTS}}\right)_{j\in[n]}\right)$ is generated using $\mathsf{DVTS.DKgen}(1^\lambda, t, n)$. Here we have, $inst_i := \left(vk^{\mathsf{DVTS}}, \left(vk_j^{\mathsf{DVTS}}\right)_{j\in[n]}, m^*\right)$ and $wit_i := v_i^{\mathsf{DVTS}}$. Additionally, we require the key structure of the VNE scheme to be as follows: The decryption key dk is the discrete log of the encryption key ek with respect to the generator g_p, i.e., $ek = g_p^{dk}$, as shown in Figure 5.

On a high level, S_i samples a key pair (ek, dk) for the VNE scheme and encrypts the witness $wit_i := v_i^{\mathsf{DVTS}}$ into the ciphertext ct. Party A ensures the ciphertext is valid, and generates an adaptor pre-signature $\tilde{\sigma}_{A,j}$ on the payment transaction $tx^j_{\mathsf{pay},i}$ with Y as the adaptor statement. Here $Y := ek$, thereby implicitly setting $y := dk$ as the adaptor witness. Party A sends the pre-signature to server S_i which checks if the adaptor statement Y is indeed the encryption key ek, and that the pre-signature is valid. If so, the server adapts the pre-signature using dk and obtains the signature σ_A on the payment transaction. The server publishes the transaction and the signatures $\sigma_{A,j}, \sigma_{S,j}$ and $\sigma_{\mathsf{aux},i}$ on the blockchain to get paid. On the other hand, party A extracts the adaptor witness y or the decryption key dk from the signature $\sigma_{A,j}$, and obtains v_i^{DVTS}. The following theorem states the security of VITĀRITand the formal proof is deferred to Section D.

Theorem 2. Let AS be a secure adaptor signature scheme with respect to a strongly unforgeable signature scheme DS and a hard dlog relation $\mathcal{R}$. Let VNE be a simulatable and sound verifiable non-committing encryption scheme and let DVTS be a secure distributed verifiable threshold service. Then VITĀRIT protocol described in Figure 7, UC-realizes the functionality $\mathcal{F}_{\mathsf{swap}}$.

We instantiate the DVTS with a Threshold VRF service (see Appendix 8); we then have the following theorem.

Theorem 3. Let $\mathsf{PKE}^{\mathsf{nc}}$ be the secure non-committing public key encryption presented in Figure 5. Let $(\mathsf{Setup}_{\mathcal{L}'}, \mathsf{Prove}_{\mathcal{L}'}, \mathsf{Vf}_{\mathcal{L}'})$ be a secure NIZK proof system for the language $\mathcal{L}'$. The construction described in Figure 9 is a secure VNE scheme with respect to the relation R_i under the decisional Diffie-Hellman assumption in groups $\mathbb{G}_q$, $\mathbb{G}_p$ and in the random oracle model.

8. Instantiation With a Threshold VRF Service

In this section, we describe instantiations of the different cryptographic tools we use in our VITĀRIT protocol. Firstly, we instantiate the signature scheme DS with the Schnorr signature scheme [56] which is set to be in the group $\mathbb{G}_p$, with generator g_p and order p. Below we discuss in detail about our instantiation of DVTS and how to correspondingly instantiate the VNE scheme.

8.1. Distributed Verifiable Random Function as DVTS

We instantiate DVTS with the (non-interative) distributed verifiable random function DVRF scheme from [9]. The interfaces are similar in both primitives and the notations are translated by replacing DVTS with DVRF. The concrete DVRF scheme that we use from [9] is based on a modified version of BLS signature scheme. Concretely, the partial evaluation $\mathsf{DVRF.PartEval}(sk_i^{\mathsf{DVRF}}, m^*)$ run by server S_i outputs $(v_i^{\mathsf{DVRF}}, \pi_i^{\mathsf{DVRF}})$, where v_i^{DVRF} is a BLS signature under the key $sk_i^{\mathsf{DVRF}} \in \mathbb{Z}_q^*$. π_i^{DVRF} is a discrete-log equality proof between the partial verification key vk_i^{DVRF} and the BLS signature $v_i^{\mathsf{DVRF}} = \mathsf{H}_q(m^*)^{sk_i^{\mathsf{DVRF}}}$, where hash function H_q is defined as $H_q : \{0,1\}^* \to \mathbb{G}_q$. Independently of the partially evaluated values, we still need a bilinear pairing function to allow for aggregation of the partial evaluations and public verifiability of the full evaluation. Thus, we instantiate the DVRF in groups with an admissible bilinear pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where $\mathbb{G}_1, \mathbb{G}_2$, and $\mathbb{G}_T$ are different groups of order q. From [9], we can see that the final VRF value $v^{\mathsf{DVRF}} \in \mathbb{G}_1$ and the VRF verification key $vk^{\mathsf{DVRF}} \in \mathbb{G}_2$. We will instantiate $\mathbb{G}_1 := \mathbb{G}_q$. Note that we we use g_q, $\hat{g}_q$, and g_T to respectively denote the generators of groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$.

Instantiation of VNE. With the instantiation of DVTS with DVRF, we have concrete relation R for the VNE scheme. Recall that in our VITĀRIT, for the VNE usage in Section 7.2.1, we required the relation R_i for $i \in [n]$. Below we specify the relation R_i, given the instantiation with DVRF for DVTS:

$$R_i := \left\{ \begin{matrix} \left(\left(vk^{\mathsf{DVRF}}, \left(vk_\ell^{\mathsf{DVRF}}\right)_{\ell\in[n]}, m^*\right), v_i^{\mathsf{DVRF}}\right) : \ s.t.\ , \\ (v_i^{\mathsf{DVRF}}, \cdot) \leftarrow \mathsf{DVRF.PartEval}(sk_i^{\mathsf{DVRF}}, m^*) \end{matrix} \right\}$$

where $\left(vk^{\mathsf{DVRF}}, \left(vk_\ell^{\mathsf{DVRF}}, sk_\ell^{\mathsf{DVRF}}\right)_{\ell\in[n]}\right)$ is generated honestly by running $\mathsf{DVRF.DKgen}(1^\lambda, t, n)$

The only component left to be instantiated in the VNE scheme from Section 7.1 is the NIZK proof for the language $\mathcal{L}$. We can instantiate the NIZK proof using ZK-SNARK [55], where we use SHA-256 for the hash function H_m needed in $\mathsf{PKE}^{\mathsf{nc}}$ and interpret the same as an arithmetic circuit.

Global input: $(tx_{\mathsf{aux},i}, pk_{\mathsf{aux},i}, \epsilon)_{i\in[n]}, (pk_{A,j}, pk_{S,j})_{j\in[t+1]}, (x \cdot t, pk_A)$,. The DVTS verification keys of servers denoted by $(vk_1^{\mathsf{DVTS}}, \ldots, vk_n^{\mathsf{DVTS}})$ and the global DVTS verification key vk^{DVTS} are also available.
Party A's input: $(sk_A, sk_{A,1}, \ldots, sk_{A,t+1})$.
Server S_i's input: $(sk_i^{\mathsf{DVTS}}, sk_{\mathsf{aux},i}, sk_{S,1}, \ldots, sk_{S,t+1})$.

Setup Phase

1) Party A generates the setup transaction
$tx_{\mathsf{stp}} := tx\left([(pk_A, \phi_A, x \cdot (t+1))], [(pk_{A,1}, pk_{S,1}, \phi_1, x) \ldots, (pk_{A,t+1}, pk_{S,t+1}, \phi_{t+1}, x)]\right)$
2) Party A publishes tx_{stp} on the blockchain by adding its signature on the transaction onto the blockchain.

Payment Phase

Server S_i for $i \in [n]$ interacts with party A who has a chosen message $m^* \leftarrow \{0,1\}^*$, and the following steps are executed:

1) Party A and server S_i generate payment transactions
$tx^j_{\mathsf{pay},i} := tx\left([(pk_{A,j}, pk_{S,j}, \phi_j, x), (pk_{\mathsf{aux},i}, \phi_{\mathsf{aux},i}, \epsilon)], [pk_i, \phi_{S,i}, x+\epsilon]\right)$ for $j \in [t+1]$. Here the public key pk_i belongs to the server S_i.
2) Server S_i generates $(v_i^{\mathsf{DVTS}}, \pi_i^{\mathsf{DVTS}}) \leftarrow \mathsf{DVTS.PartEval}(sk_i^{\mathsf{DVTS}}, m^*)$.
3) Party A together with server S_i executes the sub-protocol $\Gamma^{\mathsf{2PC}}_{\mathsf{DVTS,DS}}(inp_A, inp_{S,i}, cinp)$ for $j \in [t+1]$ (described in Figure 8), where the common input for both parties is set as $cinp := (pk_{A,j}, vk^{\mathsf{DVTS}}, vk_i^{\mathsf{DVTS}}, m^*, tx^j_{\mathsf{pay},i})$. Here party A's private input is denoted by $inp_A := sk_{A,j}$, and the private input of server S_i is denoted by $inp_{S,i} := (sk_i^{\mathsf{DVTS}}, v_i^{\mathsf{DVTS}}, \pi_i^{\mathsf{DVTS}}, sk_{S,j}, sk_{\mathsf{aux},i})$.
4) At the end of the sub-protocol, S_i gets the payment of x coins from $tx^j_{\mathsf{pay},i}$ for exactly one $j \in [t+1]$ if j-th output address in tx_{Setup} was unspent. While party A obtains v_i^{DVTS}.

Reconstruction Phase

After extracting output shares from $t+1$ servers, w.l.o.g., the first t servers, party A executes
$(v^{\mathsf{DVTS}}, \pi^{\mathsf{DVTS}}) \leftarrow \mathsf{DVTS.Combine}\left(vk^{\mathsf{DVTS}}, (vk_i^{\mathsf{DVTS}})_{i\in[n]}, m^*, \{(1, v_1^{\mathsf{DVTS}}, \pi_1^{\mathsf{DVTS}}), \ldots, (t+1, v_{t+1}^{\mathsf{DVTS}}, \pi_{t+1}^{\mathsf{DVTS}})\}\right)$. It returns v^{DVTS} as its output.

Figure 7: VITĀRIT protocol

However, as discussed earlier, given H_m is modeled as a random oracle in the security analysis, we want to avoid proofs that treat the hash function H_m as a concrete function. Therefore, we design a new cut-and-choose-based NIZK proof and a corresponding VNE scheme, which we describe in Figure 9. On a high level, the encryption algorithm takes an instance $inst$ consisting of the DVRF verification key v^{DVRF}, the partial verification keys vk_ℓ^{DVRF}, and message m^*. The witness wit is the partial evaluation value v_i^{DVRF}, and the auxiliary information is the partial secret key sk_i^{DVRF}. For $j \in [2\lambda_s]$, the algorithm samples values r_j, a_j, b_j and computes $A_j := \hat{g}_q^{a_j}$, $B_j := \hat{g}_q^{b_j}$, and the PKE$^{\mathsf{nc}}$ ciphertext ct_j that encrypts $A_j^{b_j}$ using r_j as the encryption randomness. The challenge index set J is computes using the hash function $H_c : \{0,1\}^* \to J$ where $J \subset [2\lambda_s]$ and $|J| = \lambda_s$. For indices not in J, the algorithm reveals values in plain, while for the indices in J, the algorithm reveals a value Z_j that hides v_i^{DVRF} using $A_j^{b_j}$. The algorithm also adds NIZK proofs for language $\mathcal{L}'$ that assure that values Z_j are well-formed. We specify the language below:

$$\mathcal{L}' := \left\{ \begin{array}{l} \left(i, A, B, Z, vk^{\mathsf{DVRF}}, (vk_j^{\mathsf{DVRF}})_{j\in[n]}, m^*\right) : \\ \exists (b, sk_i^{\mathsf{DVRF}}, v_i^{\mathsf{DVRF}}),\ s.t.\ , \\ B = g_q^b, \wedge\ Z = A^b \cdot v_i^{\mathsf{DVRF}} \wedge \\ (v_i^{\mathsf{DVRF}}, \cdot) \leftarrow \mathsf{DVRF.PartEval}(sk_i^{\mathsf{DVRF}}, m^*) \end{array} \right\}$$

where $\left(vk^{\mathsf{DVRF}}, \left(vk_\ell^{\mathsf{DVRF}}, sk_\ell^{\mathsf{DVRF}}\right)_{\ell\in[n]}\right)$ is honestly generated using $\mathsf{DVRF.DKgen}(1^\lambda)$. The final VNE ciphertext is the collection of all PKE$^{\mathsf{nc}}$ ciphertexts, all group elements A_j and B_j, along with all opened and unopened values.

To verify the ciphertext, the algorithm checks the well-formedness of the opened values, and for the unopened values, it checks all the NIZK proofs for the language $\mathcal{L}'$. The soundness guarantee of the cut-and-choose is that if all the checks are verified successfully, then there exists at least one unopened PKE$^{\mathsf{nc}}$ ciphertext that is correctly formed. The decryption algorithm given the decryption key dk, decrypts this well-formed ciphertext denoted by ct_k where index $k \in J$.

8.2. Optimizations

In this section, we discuss additional techniques that can improve the communication and computation of our protocol.

8.2.1. Batching in our VNE instantiation. If party A, and server S_i interact to exchange multiple partial DVRF evaluations for payments, they have to run the 2-party sub-protocol from Figure 8 as many times. If we have N number of exchanges, that is, we have instances $(inst_1, \ldots, inst_N)$ for which the witnesses will be encrypted using VNE, then this would naively involve N number of VNE encryptions and verifications. Concretely, the total number of PKE$^{\mathsf{nc}}$ ciphertexts sent from the server to party A would be $2N \cdot \lambda_s$, where λ_s is the cut-and-choose security parameter. We ignore the additional group elements here for ease of understanding.

We make use of VNE scheme from Figure 6, which has the encryption-decryption key structure from Figure 5.
Common input: $pk_{A,j}, vk^{\mathsf{DVTS}}, vk_i^{\mathsf{DVTS}}, m^*, tx^j_{\mathsf{pay},i}$

1 :	**Party** $A(inp_A)$		**Server** $S_i(inp_{S,i})$
2 :	$inp_A := sk_{A,j}$		$inp_{S,i} := (sk_i^{\mathsf{DVTS}}, v_i^{\mathsf{DVTS}}, \pi_i^{\mathsf{DVTS}}, sk_{S,j}, sk_{\mathsf{aux},i})$
3 :			$(ek, dk) \leftarrow \mathsf{VNE.KGen}(1^\lambda)$
4 :			$inst := \left(vk^{\mathsf{DVTS}}, \left(vk_j^{\mathsf{DVTS}}\right)_{j\in[n]}, m^*\right)$
5 :			$wit := v_i^{\mathsf{DVTS}}$
6 :			$z := sk_i^{\mathsf{DVTS}}$
7 :			$ct \leftarrow \mathsf{VNE.Enc}(ek, inst, wit, z)$
8 :		$\xleftarrow{(ek, ct, inst)}$	
9 :	**Check** $\mathsf{VNE.VfEnc}(ek, inst, ct) \overset{?}{=} 1$		
10 :	$Y := ek$		
11 :	$\tilde{\sigma}_{A,j} \leftarrow \mathsf{AS.pSign}(sk_{A,j}, tx^j_{\mathsf{pay},i}, Y)$		
12 :		$\xrightarrow{(Y, \tilde{\sigma}_{A,j})}$	
13 :			**Check** $Y \overset{?}{=} ek$ and $\mathsf{AS.pVf}(pk_{A,j}, tx^j_{\mathsf{pay},i}, \tilde{\sigma}_A, Y) \overset{?}{=} 1$
14 :			$\sigma_{A,j} \leftarrow \mathsf{AS.Adapt}(\tilde{\sigma}_{A,j}, y)$, where $y = dk$
15 :			$\sigma_{S,j} \leftarrow \mathsf{DS.Sign}(sk_{S,j}, tx^j_{\mathsf{aux},i})$
16 :			$\sigma_{\mathsf{aux},i} \leftarrow \mathsf{DS.Sign}(sk_{\mathsf{aux},i}, tx^j_{\mathsf{pay},i})$
17 :			**Publish** $(tx^j_{\mathsf{pay},i}, \sigma_{A,j}, \sigma_{\mathsf{aux},i})$ on the blockchain
18 :	$dk := y \leftarrow \mathsf{AS.Ext}(\tilde{\sigma}_{A,j}, \sigma_{A,j})$		
19 :	$v_i^{\mathsf{DVTS}} \leftarrow \mathsf{VNE.Dec}(dk, inst, ct)$		
20 :	**return** v_i^{DVTS}		**return** $(tx^j_{\mathsf{pay},i}, \sigma_{A,j}, \sigma_{\mathsf{aux},i})$

Figure 8: Description of $\Gamma^{\mathsf{2PC}}_{\mathsf{DVTS,DS}}(inp_A, inp_{S,i}, cinp)$

However, we propose a batching technique for our cut-and-choose argument inspired by the works of Lindell and Riva [42] and Madathil et al. [24], to amortize the cost to $2N \cdot \alpha$ ciphertexts for some constant α smaller than λ_s. More concretely, we will have a total of $2N + \frac{2N\lambda_s}{1+\log N}$ ciphertexts which is smaller than $2N \cdot \lambda_s$ whenever $\log N > \frac{2\lambda_s}{\lambda_2 - 2} - 1$. Below will sketch the high-level idea of the batching technique.

We have N instances $(inst_1, \ldots, inst_N)$, and $\mathsf{PKE}^{\mathsf{nc}}$ encryption keys $(ek_1, \ldots, ek_N)$. We want that the encryption of VNE encrypt wit_i that can be decrypted using dk_i. The high-level idea is to generate a total of $2N \cdot B$ number of $\mathsf{PKE}^{\mathsf{nc}}$ ciphertexts (and corresponding group elements), where $B = 1 + \frac{\lambda_s}{1+\log N}$. Each of these ciphertexts is with respect to an encryption key ek^*. We apply the hash function H_c which returns $N \cdot B$ (exactly half) the number of indices to be opened and the rest to be left unopened. The unopened values are randomly mapped into buckets that each can contain B entries. We have a total of N number of buckets with the j-th bucket associated with the j-th instance. To do this, the hash function H_c additionally returns the above random bucket mapping.

For $\mathsf{PKE}^{\mathsf{nc}}$ ciphertexts that are mapped to the j-th bucket, the encryption algorithm generates new $\mathsf{PKE}^{\mathsf{nc}}$ ciphertexts encrypting the same message but this time under the encryption key ek_j. Additional Z values corresponding to the unopened indices are generated in such a way that for values in j-th bucket, the corresponding Z value masks wit_j. The guarantee of the cut-and-choose now states that if the all the opened values and unopened values are correct then with an overwhelming probability there is at least one well-formed entry in each bucket. That is, one $\mathsf{PKE}^{\mathsf{nc}}$ ciphertext and corresponding Z value in each bucket are well-formed and correctly decryptable. Given the decrytion key dk_j, the decryption algorithm picks the well-formed $\mathsf{PKE}^{\mathsf{nc}}$ ciphertext and masked entry from the j-th bucket, and obtains wit_j.

8.2.2. Amortize computation in Vitārit.

Notice that in step (3) of the payment phase, party A and server S_i execute the interactive sub-protocol $\Gamma^{\mathsf{2PC}}_{\mathsf{DVTS,DS}}$ iteratively $t + 1$ times. We can amortize computation costs of these iterations by letting the server S_i execute steps 1-7 and party A execute steps 8-10 of $\Gamma^{\mathsf{2PC}}_{\mathsf{DVTS,DS}}$ in Figure 8 exactly once. The only difference between the iterations is the payment transaction and the adaptor signature generated on the transaction. Therefore, the steps between 1-10 which are independent of the transaction and adaptor signature

We make use of $\mathsf{PKE}^{\mathsf{nc}}$ scheme from Figure 5. $crs_{\mathcal{L}'}$ is available as public parameter to all algorithms.

$\underline{\mathsf{KGen}(1^\lambda)\text{: The key generation algorithm does the following:}}$

- Sample $(ek, dk) \leftarrow \mathsf{PKE}^{\mathsf{nc}}.\mathsf{KGen}(1^\lambda)$
- Return (ek, dk)

$\underline{\mathsf{Enc}(ek, inst, wit, z)\text{: The encryption algorithm does the following:}}$

- Parse $inst := \left(vk^{\mathsf{DVRF}}, \left(vk_\ell^{\mathsf{DVRF}}\right)_{\ell\in[n]}, m^*\right)$, $wit := v_i^{\mathsf{DVRF}}$ and $z := sk_i^{\mathsf{DVRF}}$
- Initialize $\mathcal{S}_{\mathsf{op}} := \mathcal{S}_{\mathsf{unop}} := \emptyset$
- For $j \in [2\lambda_s]$:
  - Sample $r_j \leftarrow \mathbb{Z}_p$, $a_j, b_j \leftarrow \mathbb{Z}_q$. Compute $A_j := g_q^{a_j}, B_j := g_q^{b_j}$
  - $ct_j \leftarrow \mathsf{PKE}^{\mathsf{nc}}.\mathsf{Enc}(ek, A_j^{b_j}; r_j)$
- Compute $J := H_c\left((A_j, B_j, ct_j)_{j\in[\lambda_s]}\right)$
- For $j \in [2\lambda_s] \setminus J$:
  - Set $\mathcal{S}_{\mathsf{op}} := \mathcal{S}_{\mathsf{op}} \cup \{(a_j, b_j, r_j)\}$
- For $j \in J$:
  - Compute $Z_j := A_j^{b_j} \cdot v_i^{\mathsf{DVRF}}$
  - Set $stmt_j := \left(i, A_j, B_j, Z_j, vk^{\mathsf{DVRF}}, \left(vk_\ell^{\mathsf{DVRF}}\right)_{\ell\in[n]}, m^*\right)$
  - Set $wit_j := (b_j, sk_i^{\mathsf{DVRF}}, v_i^{\mathsf{DVRF}})$
  - Generate proof $\pi_j \leftarrow \mathsf{Prove}_{\mathcal{L}'}(crs_{\mathcal{L}'}, stmt_j, wit_j)$
  - Set $\mathcal{S}_{\mathsf{unop}} := \mathcal{S}_{\mathsf{unop}} \cup \{(Z_j, stmt_j, \pi_j)\}$
- Set $ct := \left((ct_j, A_j, B_j)_{j\in[2\lambda_s]}, \mathcal{S}_{\mathsf{op}}, \mathcal{S}_{\mathsf{unop}}\right)$
- Return ct

$\underline{\mathsf{VfEnc}(ek, inst, ct)\text{: The verification algorithm does the following:}}$

- Parse $ct := \left((ct_j, A_j, B_j)_{j\in[2\lambda_s]}, \mathcal{S}_{\mathsf{op}}, \mathcal{S}_{\mathsf{unop}}\right)$
- Compute $J := H_c\left((A_j, B_j, ct_j)_{j\in[2\lambda_s]}\right)$
- For $j \in [2\lambda_s] \setminus J$:
  - Retrieve (a_j, b_j, r_j) from $\mathcal{S}_{\mathsf{op}}$
  - Return 0 if any of the following fail
    * Check $A_j \overset{?}{=} g_q^{a_j}$. Check $B_j \overset{?}{=} g_q^{b_j}$
    * Check $ct_j \overset{?}{=} \mathsf{PKE}^{\mathsf{nc}}.\mathsf{Enc}(ek, A_j^{b_j}; r_j)$
- For $j \in J$:
  - Retrieve $(Z_j, stmt_j, \pi_j)$ from $\mathcal{S}_{\mathsf{unop}}$
  - Return 0 if $\mathsf{Vf}_{\mathcal{L}'}(crs_{\mathcal{L}'}, stmt_j, \pi_j) = 0$
- If 0 was not returned before, return 1

$\underline{\mathsf{Dec}(dk, inst, ct)\text{: The decryption algorithm does the following:}}$

- Parse $ct := \left((ct_j, A_j, B_j)_{j\in[2\lambda_s]}, \mathcal{S}_{\mathsf{op}}, \mathcal{S}_{\mathsf{unop}}\right)$
- Compute $J := H_c\left((A_j, B_j, ct_j)_{j\in[2\lambda_s]}\right)$
- By the guarantee of cut-and-choose, there exists $k \in J$, such that $C_k \leftarrow \mathsf{PKE}^{\mathsf{nc}}.\mathsf{Dec}(dk, ct_k)$.
- Compute $wit := Z_k \cdot C_k^{-1}$
- Return wit

Figure 9: Our concrete VNE scheme.

generation can be executed exactly once instead of $t+1$ times.

8.3. Discussions

We will now briefly discuss certain interesting extensions to VITĀRIT.

Output Private DVRF. Notice that in the DVRF threshold service case, a set of $t+1$ servers can come together and compute the final VRF value v^{DVRF} that the client obtained. The client may wish to have added privacy where the final VRF value is not revealed to the service as studied in [57]. We can extend VITĀRIT DVRF instantiation to realize this by letting the client reveal a masked / blinded request $\tilde{m} = H_q(m^*)^\beta$, where $\beta \leftarrow \mathbb{Z}_q$ for some λ bit prime q and H_q is a hash function. Here the message m^* is the original message request on which the client wants a VRF to be computed. To compute the masked partial evaluation, the i-th server simply compute $\tilde{v}_i^{\mathsf{DVRF}} := (\tilde{m})^{sk_i^{\mathsf{DVRF}}}$, and the protocol follows as before. Once the client obtains the masked partial evaluation $\tilde{v}_i^{\mathsf{DVRF}}$, it can unmask and compute the original partial evaluation as: $v_i^{\mathsf{DVRF}} := \left(\tilde{v}_i^{\mathsf{DVRF}}\right)^{\beta^{-1}}$. As before, with $t+1$ partial evaluations, the client obtains the final original VRF output. However, as shown in [57], the final VRF value is computationally hidden from the servers.

Threshold Oracle Service. In the case of threshold oracle service, the client seeks $t+1$ attestations on the message m^*. VITĀRIT can be instantiated for this case as well since our protocol is described generically (Section 7) for any threshold service. If the oracles attest the message by signing the message like in [24], then we require the VNE to be instantiated with respect to the following relation:

$$R := \left\{ \begin{matrix} \left(\left(vk^{\mathsf{DS}}, m^*\right), \sigma^{\mathsf{DS}}\right) : \ s.t. \ , \\ \sigma^{\mathsf{DS}} \leftarrow \mathsf{DS}.\mathsf{Sign}(sk^{\mathsf{DS}}, m^*) \end{matrix} \right\}$$

where DS is the signature scheme, vk^{DS} is the signature verification key, sk^{DS} is the corresponding signing key, and σ^{DS} is the signature. If the signature scheme used is BLS signatures [47], then given the structural similarities between the signature scheme and the DVRF instantiation [9] we can use our concrete instantiation from Section 8 with only minimal changes.

Paper Organization. In Section 2 we discuss some of the related works in the literature and in Section 4 we briefly introduce the mathematical notations and required technical background needed for the paper. We introduce the UC-based formal model for VITĀRIT in Section 5 followed the VITĀRIT protocol description in Section 7. In Section 9 we present a practical performance evaluation of VITĀRIT. Formal proofs of VITĀRIT and its core component are deferred to Section D and Section C, respectively. Concrete instantiations of the building blocks used in VITĀRIT are discussed in Section 8.

9. Performance Analysis

We implement [6]and benchmark the VNE (Figure 9) and the VITĀRIT protocol using Rust cryptographic libraries `derek_25519, blst and bls12_381` crates on MAX OSX with $16GB$ RAM. All the reported timings are mean values taken over 50 instances of the protocol run. Each server uses a BLS signature-based threshold VRF on the curve `BLS12_381` curve to generate a partial VRF output and participates with the client in a two-party exchange protocol for the payment information. The proof of correct evaluation is a Schnorr protocol-based zero-knowledge proof. In the encryption scheme $\mathsf{PKE}^{\mathsf{nc}}$ of Figure 5, we choose p to be a 256 bit value from the `Ed25519` curve and the message length to be 48 byte which is an output of the VRF function. The hash $H_m(\cdot)$ maps to 384 a bit value. This optimization reduces the size of the 3-tuple ciphertext in which, now two elements are 256 bits.

While encrypting the partial VRF output using the VNE scheme, each server samples 64 random values and uses them to generate ciphertexts for the $\mathsf{PKE}^{\mathsf{nc}}$ encryption. Later we vary the number of values to compare the time taken for different cut-and-choose parameters. The VRF output is an element on the BLS12-381 curve, which is the witness for the VNE encryption. The server VNE-encrypts the partial output, which also includes proof of the correctness of the encryption. After verifying the VNE encryption, the client generates a pre-signature. The server verifies and adapts the pre-signature to obtain the payment transaction and signature on the transaction. The timings for each of the steps in the two-party protocol are provided in Table 1.

The proof of the correctness of the VNE involves a cut-and-choose method. We choose the security parameter $2\lambda_s$ to be 64. Half the total number of encryptions are opened and half of them are masked and forwarded. The indices of the encryptions that are opened are generated as a hash output. For the unopened encryptions, the partial evaluation is masked by the random value; the server proves in zero-knowledge that the value has been generated correctly, and the masked value is indeed the partial evaluation of the server. The server uses a schnorr-protocol-based zero-knowledge proof (see E for the exact proof) to prove the correctness of the masked value.

Communication complexity. The VRF computed is a BLS-signature-based VRF function with an output of size 48 bytes. $\mathsf{PKE}^{\mathsf{nc}}$ encryption is used to encrypt the VRF output, and each encryption instance has a ciphertext of 896 bits. The server VNE encrypts the VRF output, which utilizes the PKE encryption. In the VNE-based cut-and-choose method, the server forwards 64 encryptions, of which 32 random encryptions are opened. Thus, the total ciphertext size is $64 * 896 = 57.3$kb; the opened set involves forwarding three values per $\mathsf{PKE}^{\mathsf{nc}}$ encryption amounting to 1152 bits per $\mathsf{PKE}^{\mathsf{nc}}$. For the unopened set, the server forwards the masked value and the proof of correctness, constituting 5 elements, each of size 384 bits, totaling 1.92kb. Upon successful verification, the client forwards a pre-signature and public key amounting to 512 bits. After adapting the signature, the server publishes a transaction of size up to ~ 218 bytes on the blockchain. Refer Table 2 object sizes.

6. https://github.com/easwarvivek/vitarit

TABLE 1. TIME TAKEN FOR THE CLIENT AND SERVER IN THE TWO-PARTY PROTOCOL FOR PAYMENT AND PARTIAL VALUE EXCHANGE

Client	Time (msec)
$\mathsf{PKE}^{\mathsf{nc}}$ Decryption	491μ sec
ZKP Verification of unopened $\mathsf{PKE}^{\mathsf{nc}}$	1.48 mec
Pre-Signature	187μsec
Server	**Time (msec)**
VRF generation	340μ sec
$\mathsf{PKE}^{\mathsf{nc}}$ Encryption	1.98 msec
ZKP generation of unopened $\mathsf{PKE}^{\mathsf{nc}}$	1.67 msec
Adaptor Verification	179μsec
Signature generation	186μsec

TABLE 2. SIZE OF EACH OF THE ELEMENTS OF THE PROTOCOL

Element	Size
Partial VRF output	382 bits
$\mathsf{PKE}^{\mathsf{nc}}$ ciphertext	896 bits
ZKP of unopened $\mathsf{PKE}^{\mathsf{nc}}$	1536 bits
Adaptor Pre-signature	256 bits

Computation complexity. The VRF partial evaluation involves a hash function computation and exponentiation on the `BLS12_381` curve and takes 340μ sec. Each $\mathsf{PKE}^{\mathsf{nc}}$ encryption involves sampling two random values, 4 exponentiation on elements of size 256 bits and takes 1.98 msec. The $\mathsf{PKE}^{\mathsf{nc}}$ decryption involves one exponentiation along with other operations and takes 491μ sec. The VNE encryption involves generating multiple (64) $\mathsf{PKE}^{\mathsf{nc}}$ ciphertexts and takes a total of 127 msec. The proof of correctness of Z_j value, a Schnorr based proof, takes 1.98 msec for each unopened $\mathsf{PKE}^{\mathsf{nc}}$. Thus, takes a total of 63.36 msec sec for the VNE instance. The client encrypts the values for the opened instances and checks against the ciphertexts. The client checks the proof of correctness for the unopened instances, taking 1.48 msec for each.

We vary the cut-and-choose parameter and compute the time the server takes (see Figure 10).

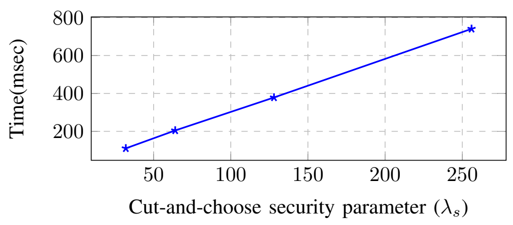

Figure 10: Server time in the two-party protocol for varying cut-and-choose parameter λ_s

Improving smart contract based schemes. To demonstrate that VITĀRIT improves even a distributed VRF realized through smart contracts (gas cost-wise), we deploy smart contracts on Ethereum with and without using VITĀRIT and compare the gas costs. In the traditional approach, each party submits the partial VRF output to the smart contract which verifies the correctness of the value. Once the threshold number of values are verified, they are aggregated and the final value is verified using a pairing based check (https://holesky.etherscan.io/tx/0xd6c9bbce9fbd6c892ef7c52757b646d860dc67e8e8f131f5c4fc9734cee74d14). Before the start of the operations, the servers register their public key with the smart contract. For a fair comparison, we consider the servers to get paid as soon as the final VRF value is verified. Since the VRF verification keys need to be stored in a specific format, it takes 502K gas to store 5 public keys for a five server system. The deployment of smart contract takes 1891K gas since it includes storing variables for BLS verification precompiles etc. It takes 219K gas for one BLS pairing verification after receiving and checking the input. This approach also incurs costs to store the proof of correctness of the values.

To realize VITĀRIT in Ethereum, the simplest way is to realize the protocol as is. In VITĀRIT , the auxiliary address allows each server to obtain the payment for the service only once. The address holds a one-time spendable value; this is realized in Ethereum using a smart contract with deposit which can used only once for an input. After the interaction with the client, each server posts one payment transaction (https://holesky.etherscan.io/tx/0x1182b91846e465c9dd9a2a1a9b8d1b2f847b24bc37c2f2e943da99b93f7ef2a4). The gas cost per server for obtaining the payment is 21K since it is a simple payment transaction. For registering the public keys and the auxiliary deposit, it takes 227K gas for deploying the smart contract and registering the public keys together. This completely avoids high smart contract deployment cost, Schnorr verification proofs of partial VRF outputs, pairing-based verification of the final VRF output and the storage costs of the proofs and partial values.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under grant CNS1846316.

References

[1] "Solidity, ethereum." https://docs.soliditylang.org/en/v0.8.23/.

[2] "Chainlink vrf: Explanation and alternatives." https://supraoracles.com/academy/chainlink-vrf/.

[3] A. Bhat, N. Shrestha, Z. Luo, A. Kate, and K. Nayak, "RandPiper - reconfiguration-friendly random beacons with quadratic communication," pp. 3502–3524, ACM Press, 2021.

[4] S. A. K. Thyagarajan, G. Castagnos, F. Laguillaumie, and G. Malavolta, "Efficient CCA timed commitments in class groups," pp. 2663–2684, ACM Press, 2021.

[5] S. Das, V. Krishnan, I. M. Isaac, and L. Ren, "Spurt: Scalable distributed randomness beacon with transparent setup," pp. 2502–2517, IEEE Computer Society Press, 2022.

[6] K. Choi, A. Manoj, and J. Bonneau, "SoK: Distributed randomness beacons," pp. 75–92, IEEE Computer Society Press, 2023.

[7] S. Micali, M. O. Rabin, and S. P. Vadhan, "Verifiable random functions," in *40th FOCS*, (New York, NY, USA), pp. 120–130, IEEE Computer Society Press, Oct. 17–19, 1999.

[8] "Timelock encryption/decryption made practical." https://github.com/drand/.

[9] D. Galindo, J. Liu, M. Ordean, and J.-M. Wong, "Fully distributed verifiable random functions and their application to decentralised random beacons." Cryptology ePrint Archive, Report 2020/096, 2020. https://eprint.iacr.org/2020/096.

[10] "Supra oracles." https://supraoracles.com/.

[11] B. Liu, P. Szalachowski, and J. Zhou, "A first look into defi oracles," in *IEEE International Conference on Decentralized Applications and Infrastructures, DAPPS 2021, Online Event, August 23-26, 2021*, pp. 39–48, IEEE, 2021.

[12] F. Zhang, D. Maram, H. Malvai, S. Goldfeder, and A. Juels, "Deco: Liberating web data using decentralized oracles for tls," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1919–1938, 2020.

[13] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 910–927, IEEE, 2020.

[14] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous multi-hop locks for blockchain scalability and interoperability," in *NDSS 2019*, (San Diego, CA, USA), The Internet Society, Feb. 24-27, 2019.

[15] S. A. Krishnan Thyagarajan and G. Malavolta, "Lockable signatures for blockchains: Scriptless scripts for all signatures," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 937–954, 2021.

[16] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi, "Generalized channels from limited blockchain scripts and adaptor signatures," LNCS, pp. 635–664, Springer, Heidelberg, Germany, 2021.

[17] L. Aumayr, M. Maffei, O. Ersoy, A. Erwig, S. Faust, S. Riahi, K. Hostáková, and P. Moreno-Sanchez, "Bitcoin-compatible virtual channels," pp. 901–918, IEEE Computer Society Press, 2021.

[18] L. Aumayr, P. Moreno-Sanchez, A. Kate, and M. Maffei, "Donner: UTXO-based virtual channels across multiple hops." Cryptology ePrint Archive, Report 2021/855, 2021. https://eprint.iacr.org/2021/855.

[19] E. Tairi, P. Moreno-Sanchez, and M. Maffei, "Post-quantum adaptor signature for privacy-preserving off-chain payments," LNCS, pp. 131–150, Springer, Heidelberg, Germany, 2021.

[20] E. Tairi, P. Moreno-Sanchez, and M. Maffei, "A^2L: Anonymous atomic locks for scalability in payment channel hubs," pp. 1834–1851, IEEE Computer Society Press, 2021.

[21] S. A. Thyagarajan, G. Malavolta, and P. Moreno-Sanchez, "Universal atomic swaps: Secure exchange of coins across all blockchains," in *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 1299–1316, 2022.

[22] L. Aumayr, S. A. K. Thyagarajan, G. Malavolta, P. Moreno-Sanchez, and M. Maffei, "Sleepy channels: Bi-directional payment channels without watchtowers," pp. 179–192, ACM Press, 2022.

[23] N. Glaeser, M. Maffei, G. Malavolta, P. Moreno-Sanchez, E. Tairi, and S. A. K. Thyagarajan, "Foundations of coin mixing services," pp. 1259–1273, ACM Press, 2022.

[24] V. Madathil, S. A. Thyagarajan, D. Vasilopoulos, L. Fournier, G. Malavolta, and P. Moreno-Sanchez, "Practical decentralized oracle contracts for cryptocurrencies," *Cryptology ePrint Archive*, 2022.

[25] L. Hanzlik, J. Loss, S. A. Thyagarajan, and B. Wagner, "Sweep-UC: Swapping coins privately." Cryptology ePrint Archive, Report 2022/1605, 2022. https://eprint.iacr.org/2022/1605.

[26] D. Mazieres, "The stellar consensus protocol: A federated model for internet-level consensus," *Stellar Development Foundation*, vol. 32, 2015.

[27] D. Schwartz, N. Youngs, A. Britto, *et al.*, "The ripple protocol consensus algorithm," *Ripple Labs Inc White Paper*, vol. 5, no. 8, 2014.

[28] R. W. F. Lai, V. Ronge, T. Ruffing, D. Schröder, S. A. K. Thyagarajan, and J. Wang, "Omniring: Scaling private payments without trusted setup," in *ACM CCS 2019* (L. Cavallaro, J. Kinder, X. Wang, and J. Katz, eds.), pp. 31–48, ACM Press, Nov. 11–15, 2019.

[29] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE Symposium on Security and Privacy*, (Berkeley, CA, USA), pp. 459–474, IEEE Computer Society Press, May 18–21, 2014.

[30] "Web3 should be built on bitcoin." https://www.nasdaq.com/articles/web3-should-be-built-on-bitcoin.

[31] D. community, "Specification for discreet log contracts." https://github.com/discreetlogcontracts/dlcspecs.

[32] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pp. 136–145, IEEE, 2001.

[33] I. Bentov and R. Kumaresan, "How to use bitcoin to design fair protocols," in *CRYPTO 2014, Part II* (J. A. Garay and R. Gennaro, eds.), vol. 8617 of *LNCS*, (Santa Barbara, CA, USA), pp. 421–439, Springer, Heidelberg, Germany, Aug. 17–21, 2014.

[34] Github Project. https://github.com/chatch/hashed-timelock-contract-ethereum.

[35] M. Campanelli, R. Gennaro, S. Goldfeder, and L. Nizzardo, "Zero-knowledge contingent payments revisited: Attacks and payments for services," in *ACM CCS 2017* (B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds.), (Dallas, TX, USA), pp. 229–243, ACM Press, Oct. 31 – Nov. 2, 2017.

[36] "Lightning network." https://lightning.network/.

[37] V. Goyal, Y. Song, and A. Srinivasan, "Traceable secret sharing and applications," in *CRYPTO 2021, Part III* (T. Malkin and C. Peikert, eds.), vol. 12827 of *LNCS*, (Virtual Event), pp. 718–747, Springer, Heidelberg, Germany, Aug. 16–20, 2021.

[38] D. Boneh and C. Komlo, "Threshold signatures with private accountability," in *CRYPTO 2022, Part IV*, LNCS, (Santa Barbara, CA, USA), pp. 551–581, Springer, Heidelberg, Germany, Aug. 2022.

[39] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi, "Generalized channels from limited blockchain scripts and adaptor signatures," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 635–664, Springer, 2021.

[40] M. Chase, C. Ganesh, and P. Mohassel, "Efficient zero-knowledge proof of algebraic and non-algebraic statements with applications to privacy preserving credentials," in *CRYPTO 2016, Part III* (M. Robshaw and J. Katz, eds.), vol. 9816 of *LNCS*, (Santa Barbara, CA, USA), pp. 499–530, Springer, Heidelberg, Germany, Aug. 14–18, 2016.

[41] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno, "Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation," in *2016 IEEE Symposium on Security and Privacy*, (San Jose, CA, USA), pp. 235–254, IEEE Computer Society Press, May 22–26, 2016.

[42] Y. Lindell and B. Riva, "Cut-and-choose Yao-based secure computation in the online/offline and batch settings," in *CRYPTO 2014, Part II* (J. A. Garay and R. Gennaro, eds.), vol. 8617 of *LNCS*, (Santa Barbara, CA, USA), pp. 476–494, Springer, Heidelberg, Germany, Aug. 17–21, 2014.

[43] Z. Brakerski, P. Branco, N. Döttling, S. Garg, and G. Malavolta, "Constant ciphertext-rate non-committing encryption from standard assumptions," in *TCC 2020, Part I* (R. Pass and K. Pietrzak, eds.), vol. 12550 of *LNCS*, (Durham, NC, USA), pp. 58–87, Springer, Heidelberg, Germany, Nov. 16–19, 2020.

[44] W. Dai, T. Okamoto, and G. Yamamoto, "Stronger security and generic constructions for adaptor signatures." Cryptology ePrint Archive, Report 2022/1687, 2022. https://eprint.iacr.org/2022/1687.

[45] A. De Santis, S. Micali, and G. Persiano, "Non-interactive zero-knowledge proof systems," in *Conference on the Theory and Application of Cryptographic Techniques*, pp. 52–72, Springer, 1987.

[46] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," in *EUROCRYPT'99* (J. Stern, ed.), vol. 1592 of *LNCS*, (Prague, Czech Republic), pp. 295–310, Springer, Heidelberg, Germany, May 2–6, 1999.

[47] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," in *ASIACRYPT 2001* (C. Boyd, ed.), vol. 2248 of *LNCS*, (Gold Coast, Australia), pp. 514–532, Springer, Heidelberg, Germany, Dec. 9–13, 2001.

[48] D. Boneh and M. K. Franklin, "Identity-based encryption from the Weil pairing," in *CRYPTO 2001* (J. Kilian, ed.), vol. 2139 of *LNCS*, (Santa Barbara, CA, USA), pp. 213–229, Springer, Heidelberg, Germany, Aug. 19–23, 2001.

[49] A. Cerulli, A. Connolly, G. Neven, F.-S. Preiss, and V. Shoup, "vetkeys: How a blockchain can keep many secrets." Cryptology ePrint Archive, Paper 2023/616, 2023. https://eprint.iacr.org/2023/616.

[50] L. Baird, S. Garg, A. Jain, P. Mukherjee, R. Sinha, M. Wang, and Y. Zhang, "Threshold signatures in the multiverse," pp. 1454–1470, IEEE Computer Society Press, 2023.

[51] S. Garg, A. Jain, P. Mukherjee, R. Sinha, M. Wang, and Y. Zhang, "hints: Threshold signatures with silent setup." Cryptology ePrint Archive, Paper 2023/567 (To appear in IEEE Security and Privacy – 2024), 2023. https://eprint.iacr.org/2023/567.

[52] S. Das, P. Camacho, Z. Xiang, J. Nieto, B. Bünz, and L. Ren, "Threshold signatures from inner product argument: Succinct, weighted, and multi-threshold," *IACR Cryptol. ePrint Arch.*, p. 598, 2023.

[53] S. Agrawal, P. Mohassel, P. Mukherjee, and P. Rindal, "DiSE: Distributed symmetric-key encryption," in *ACM CCS 2018* (D. Lie, M. Mannan, M. Backes, and X. Wang, eds.), (Toronto, ON, Canada), pp. 1993–2010, ACM Press, Oct. 15–19, 2018.

[54] J. B. Nielsen, "Non-committing encryption is too easy in the random oracle model," *BRICS Report Series*, vol. 8, 2001.

[55] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *2013 IEEE Symposium on Security and Privacy*, (Berkeley, CA, USA), pp. 238–252, IEEE Computer Society Press, May 19–22, 2013.

[56] C.-P. Schnorr, "Efficient signature generation by smart cards," *Journal of Cryptology*, vol. 4, pp. 161–174, Jan. 1991.

[57] A. Kate, E. V. Mangipudi, S. Maradana, and P. Mukherjee, "Flexirand: Output private (distributed) vrfs and application to blockchains," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, (New York, NY, USA), p. 1776–1790, Association for Computing Machinery, 2023.

Appendix A. Extended Preliminaries

Adaptor Signatures. We recall the missing definitions for adaptor signatures.

Definition 7 (Adaptor Signatures). An adaptor signature scheme AS w.r.t. a hard relation R and a signature scheme $\mathsf{DS} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$ has algorithms $(\mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVf}, \mathsf{Ext})$ defined as:

- $\hat{\sigma} \leftarrow \mathsf{pSign}(sk, m, Y)$: the pre-sign algorithm takes as input a signing key sk, message $m \in \{0,1\}^*$ and statement $Y \in \mathcal{L}_R$, outputs a pre-signature $\hat{\sigma}$.
- $0/1 \leftarrow \mathsf{pVf}(vk, m, Y, \hat{\sigma})$: the pre-verify algorithm takes as input a verification key vk, message $m \in \{0,1\}^*$, statement $Y \in \mathcal{L}_R$ and pre-signature $\hat{\sigma}$, outputs either 1 (for valid) or 0 (for invalid).
- $\sigma/\perp \leftarrow \mathsf{Adapt}(\hat{\sigma}, y)$: the adapt algorithm takes as input a pre-signature $\hat{\sigma}$ and witness y, outputs a signature σ or $\perp$ denoting error.
- $y \leftarrow \mathsf{Ext}(\sigma, \hat{\sigma}, Y)$: the extract algorithm takes as input a signature σ, pre-signature $\hat{\sigma}$ and statement $Y \in \mathcal{L}_R$, outputs a witness y such that $(Y, y) \in R$, or $\perp$.

Definition 8 (Pre-signature Correctness). An adaptor signature scheme AS satisfies pre-signature correctness if for every $\lambda \in \mathbb{N}$, every message $m \in \{0,1\}^*$ and every statement/witness pair $(Y, y) \in R$, the following holds:

$$\Pr\left[\begin{array}{c} \mathsf{pVf}(vk, m, Y, \hat{\sigma}) = 1 \\ \wedge \\ \mathsf{Vf}(vk, m, \sigma) = 1 \\ \wedge \\ (Y, y') \in R \end{array} \middle| \begin{array}{l} (sk, vk) \leftarrow \mathsf{KGen}(1^\lambda) \\ \hat{\sigma} \leftarrow \mathsf{pSign}(sk, m, Y) \\ \sigma := \mathsf{Adapt}(\hat{\sigma}, y) \\ y' := \mathsf{Ext}(\sigma, \hat{\sigma}, Y) \end{array}\right] = 1.$$

Next, we formally define the security properties of an adaptor signature scheme.

Definition 9 (Unforgeability). An adaptor signature scheme AS is $\mathsf{aEUF\text{-}CMA}$ secure if for every PPT adversary $\mathcal{A}$ there exists a negligible function negl such that:

$$\Pr\left[\mathsf{aSigForge}_{\mathcal{A},\mathsf{AS}}(\lambda) = 1\right] \leq \mathsf{negl}(\lambda)$$

where the experiment $\mathsf{aSigForge}_{\mathcal{A},\mathsf{AS}}$ is defined in Figure 11.

Definition 10 (Pre-signature Adaptability). An adaptor signature scheme AS satisfies pre-signature adaptability if for any $\lambda \in \mathbb{N}$, any message $m \in \{0,1\}^*$, any statement/witness pair $(Y, y) \in R$, any key pair $(sk, vk) \leftarrow \mathsf{KGen}(1^\lambda)$ and any pre-signature $\hat{\sigma} \leftarrow \{0,1\}^*$ with $\mathsf{pVf}(vk, m, Y, \hat{\sigma}) = 1$ we have:

$$\Pr[\mathsf{Vf}(vk, m, \mathsf{Adapt}(\hat{\sigma}, y)) = 1] = 1$$

```
$\mathsf{aSigForge}_{\mathcal{A},\mathsf{AS}}(\lambda)$
$\mathcal{Q} := \emptyset$
$(sk, vk) \leftarrow \mathsf{KGen}(1^\lambda)$
$m \leftarrow \mathcal{A}^{\mathsf{Sign}\mathcal{O}(\cdot), \mathsf{pSign}\mathcal{O}(\cdot,\cdot)}(vk)$
$(Y, y) \leftarrow \mathsf{GenR}(1^\lambda)$
$\hat{\sigma} \leftarrow \mathsf{pSign}(sk, m, Y)$
$\sigma \leftarrow \mathcal{A}^{\mathsf{Sign}\mathcal{O}(\cdot), \mathsf{pSign}\mathcal{O}(\cdot,\cdot)}(\hat{\sigma}, Y)$
return $(m \notin \mathcal{Q} \wedge \mathsf{Vf}(vk, m, \sigma))$

$\mathsf{Sign}\mathcal{O}(m)$
$\sigma \leftarrow \mathsf{Sign}(sk, m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\sigma$

$\mathsf{pSign}\mathcal{O}(m, Y)$
$\hat{\sigma} \leftarrow \mathsf{pSign}(sk, m, Y)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\hat{\sigma}$
```

Figure 11: Unforgeabiltiy experiment of adaptor signatures

```
$\mathsf{aWitExt}_{\mathcal{A},\mathsf{AS}}(\lambda)$
$\mathcal{Q} := \emptyset$
$(sk, vk) \leftarrow \mathsf{KGen}(1^\lambda)$
$(m, Y) \leftarrow \mathcal{A}^{\mathsf{Sign}\mathcal{O}(\cdot), \mathsf{pSign}\mathcal{O}(\cdot,\cdot)}(vk)$
$\hat{\sigma} \leftarrow \mathsf{pSign}(sk, m, Y)$
$\sigma \leftarrow \mathcal{A}^{\mathsf{Sign}\mathcal{O}(\cdot), \mathsf{pSign}\mathcal{O}(\cdot,\cdot)}(\hat{\sigma})$
$y' := \mathsf{Ext}(vk, \sigma, \hat{\sigma}, Y)$
return $(m \notin \mathcal{Q} \wedge (Y, y') \notin R$
    $\wedge\ \mathsf{Vf}(vk, m, \sigma))$

$\mathsf{Sign}\mathcal{O}(m)$
$\sigma \leftarrow \mathsf{Sign}(sk, m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\sigma$

$\mathsf{pSign}\mathcal{O}(m, Y)$
$\hat{\sigma} \leftarrow \mathsf{pSign}(sk, m, Y)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\hat{\sigma}$
```

Figure 12: Witness extractability experiment for adaptor signatures

Definition 11 (Witness Extractability). An adaptor signature scheme AS is *witness extractable* if for every PPT adversary $\mathcal{A}$, there exists a negligible function negl such that the following holds:

$$\Pr[\mathsf{aWitExt}_{\mathcal{A},\mathsf{AS}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$$

where the experiment $\mathsf{aWitExt}_{\mathcal{A},\mathsf{AS}}$ is defined in Figure 12.

Appendix B. Further definitions related to VNE

Definition 12 (Correctness). A verifiable non-committing encryption scheme $\mathsf{VNE} := (\mathsf{KGen}, \mathsf{Enc}, \mathsf{VfEnc}, \mathsf{Dec})$ with respect to relation R, is said to be correct if for all $\lambda \in \mathbb{N}$, all $(inst, wit, z) \in \mathsf{SUPP}(\mathsf{GenR})$, all $(ek, dk) \in \mathsf{KGen}(1^\lambda)$, all $ct \leftarrow \mathsf{Enc}(ek, inst, wit, z)$, it holds that:

- $\Pr[\mathsf{VfEnc}(ek, inst, ct) = 1] = 1$.
- $\Pr[(inst, \mathsf{Dec}(dk, inst, ct)) \in R] = 1$

Below we state the definition of simulatability of VNE that captures the non-committing property. On a high level, we want the adversary not to be able to distinguish two distributions, one REAL and another IDEAL. The IDEAL distribution is generated with the aid of a simulator Sim that generates ciphertexts without access to the encrypted witness wit, and later gives the adversary the decryption key dk. The adversary can clearly decrypt the ciphertext, and we want the view of the adversary to be no different now compared to when it receives an honestly generated ciphertext as in REAL.

Definition 13 (Simulatability). A VNE scheme with respect to NP relation R is said to be *simulatable* if for all $\lambda \in \mathbb{N}$, all PPT adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a simulator $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$, such that the following distributions IDEAL and REAL are computationally indistinguishable to $\mathcal{A}$, where

$$\mathsf{REAL} = \left\{ (ek, dk, inst, wit, ct) : \begin{array}{c} (ek, dk) \leftarrow \mathsf{KGen}(1^\lambda) \\ (inst, wit, z) \leftarrow \mathsf{GenR}(1^\lambda) \\ ct \leftarrow \mathsf{Enc}(pk, inst, wit, z) \\ (\mathsf{st}_0, \top) \leftarrow \mathcal{A}_1(ek, ct, inst) \\ wit \leftarrow \mathcal{A}_2(\mathsf{st}_0, dk) \end{array} \right\}$$

$$\mathsf{IDEAL} = \left\{ (ek, dk, inst, wit, ct) : \begin{array}{c} (ek, dk') \leftarrow \mathsf{KGen}(1^\lambda) \\ (inst, wit, z) \leftarrow \mathsf{GenR}(1^\lambda) \\ (\mathsf{st}'_0, ct) \leftarrow \mathsf{Sim}_1(ek, inst) \\ (\mathsf{st}_0, \top) \leftarrow \mathcal{A}_1(ek, ct, inst) \\ dk \leftarrow \mathsf{Sim}_2(\mathsf{st}'_0, dk', wit, z) \\ wit \leftarrow \mathcal{A}_2(\mathsf{st}_0, dk) \end{array} \right\}.$$

Below we state the definition of soundness for VNE.

Definition 14 (Soundness). A VNE scheme with respect to NP relation R is *sound* if for all $\lambda \in \mathbb{N}$, there exists a negligible function negl, and no PPT adversary $\mathcal{A}$ that $\mathcal{A}$ outputs $(pk, inst, ct)$, and the following holds simultaneously with more than $\mathsf{negl}(\lambda)$ probability:

- $\mathsf{VfEnc}(pk, inst, ct) = 1$
- $(inst, \mathsf{Dec}(dk, inst, ct)) \notin R$

Appendix C. Security Proofs for VNE

Proof: [Proof of Theorem 1] To see why simulatability holds, we will proceed using hybrid arguments.

Consider a distribution denoted by REAL_1, which is the same as REAL, except that the proof π for language $\mathcal{L}$ is generated using the NIZK simulation $\mathcal{S}_\mathcal{L}$. We can see that the distributions REAL and REAL_1 are computationally indistinguishable due to the zero-knowledge property of the NIZK proof system.

Now consider another distribution denoted by REAL_2, which is the same as REAL_1, except that the $\mathsf{PKE}^{\mathsf{nc}}$ ciphertext c is generated by encrypting a random value. That is, c is set as (c_1, c_2, c_3) where $c_1 := g_p^r$, $c_2 := (ek)^r \cdot g_p^s$ and $c_3 \leftarrow \{0,1\}^{|m|}$. Once the adversary $\mathcal{A}_1$ returns $\top$, the random oracle H_m is programmed at g_p^s such that $H_m(g_p^s) := c_3 \oplus wit$. Since c_3 is chosen at random and therefore $H_m(g_p^s)$ is also random, the view of the adversary is identical in both REAL_1 and REAL_2. We now argue that REAL_2 is the same as IDEAL, as we let Sim_1 generate ct exactly as in REAL_2 and Sim_2 perform the reprogramming of the random oracle as done in REAL_2. By the standard hybrid argument, we can see that REAL and IDEAL are computationally indistinguishable.

The soundness of the VNE construction holds directly from the soundness of the NIZK proof system for $\mathcal{L}$, and the perfect correctness of the $\mathsf{PKE}^{\mathsf{nc}}$ encryption scheme. ☐

Proof: [Proof of Theorem 3]

The VNE scheme presented in Figure 9 is secure if it is sound and simulatable. We begin the proof by showing the soundness of the scheme and follow with simulatability.

Soundness. Let $\mathcal{A}$ be an adversary against the soundness experiment of the non-committing public key encryption scheme $\mathsf{PKE}^{\mathsf{nc}}$, i.e., an adversary that outputs a ciphertext that cannot be decrypted to a valid witness for the relation R_i.

We will construct a reduction $\mathcal{R}$ that uses $\mathcal{A}$ as a sub-procedure and breaks the soundness of the NIZK proof for language $\mathcal{L}'$. First, the reduction receives the common reference string $crs_{\mathcal{L}'}$ from the soundness experiment and provides it to the adversary. After running the adversary $\mathcal{A}$, the reduction receives $(pk, inst, ct = ((ct_j, A_j, B_j)_{j\in[2\lambda_s]}, \mathcal{S}_{\mathsf{op}}, \mathcal{S}_{\mathsf{unop}}))$, where $\mathcal{S}_{\mathsf{unop}} = \{(Z_j, stmt_j, \pi_j\}_{j\in J}$. In the next step, it randomly picks one of the $(stmt_j, \pi_j)$ and returns it to its challenger.

We know that with overwhelming probability $(1 - \frac{1}{2^{\lambda_S}})$, for at least one $j \in J$, there exists a ciphertext $ct_j \leftarrow \mathsf{PKE}^{\mathsf{nc}}.\mathsf{Enc}(ek, A_j^{b_j}; r_j)$, i.e., it is possible to decrypt a valid $A_j^{b_j}$ where $A_j = g_q^{a_j}$ and $B_j = g_q^{b_j}$. In other words, at least one valid key will be used to decrypt Z_j to a valid share. Since the adversary $\mathcal{A}$ breaks the soundness experiment for the non-committing public key encryption scheme, it follows that Z_j does not encrypt a valid partial evaluation of the threshold service. Thus, the only way for adversary $\mathcal{A}$ to output such ciphertext ct is to create a proof π_j for a false statement $stmt_j$ (for the correctness of value Z_j).

It follows that since the reduction picks one of the $(stmt_j, \pi_j)$ with $j \in J$ at random, with probability at least $1/\lambda_S$ it will output a false statement with a valid proof, thus breaking the soundness property of the NIZK-proof system. This ends the proof.

Simulatability. We will show this proof using the game-based approach, where we make small and indistinguishable changes.

Hyb_0: The original simulatability experiment.

Hyb_1: Similar to the previous hybrid but with a slight change. We use the zero-knowledge property of the NIZK proof system for language $\mathcal{L}'$ and simulate all the proofs created as part of ciphertext ct.

Hyb_2: Similar to the previous hybrid but with a slight change. We pick a subset $J' \subset [2\lambda_s]$ and then program the random oracle H_c in a way that $J' = ((A_j, B_j, ct_j)_{j\in[\lambda_s]}) = J$.

Hyb_3: In this hybrid for all $j \in J'$ the ciphertext

$$ct_j \leftarrow \mathsf{PKE}^{\mathsf{nc}}.\mathsf{Enc}(ek, A_j^{b_j}; r_j)$$

is computed as

$$ct_j \leftarrow \mathsf{PKE}^{\mathsf{nc}}.\mathsf{Enc}(ek, g_q^{c_j}; r_j)$$

for some $c_j \leftarrow\$ \mathbb{Z}_q$. Additionally, we compute $Z_j := g_q^{c_j} \cdot v_i^{\mathsf{DVRF}}$.

Hyb_4: Similar to the previous hybrid, but we abort the experiment in case the adversary, before receiving the decryption key dk, queries the random oracle H_p (see Fig. Figure 5) for g_p^s where $ct_j = (c_1, c_2, c_3)$, $c_1 = g_p^r$ and $c_2 = (ek)^r \cdot g_p^s$ for one $j \in J'$.

Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ be an adversary distinguishing the distributions REAL and IDEAL. Moreover, let $D = (ek, dk, inst, wit, ct)$ be an instance of one of the distribution and let us denote advantage of $\mathcal{A}$ as:

$$\begin{aligned}\mathbf{Adv}_{\mathsf{sim}} = |&\Pr[\mathcal{A}(D) \Rightarrow 1 \mid D \in \mathsf{REAL}] \\ &- \Pr[\mathcal{A}(D) \Rightarrow 1 \mid D \in \mathsf{IDEAL}]\,|\end{aligned}$$

Claim 1. We claim that the changes made in Hyb_1 only increase the adversary's advantage by a negligible factor. In particular, it only increases by the advantage on an adversary breaking the zero-knowledge property of the NIZK proof system for language $\mathcal{L}'$.

Proof: The proof follows by a straightforward reduction that replaces all proofs with simulated ones. □

Claim 2. We claim that the changes in Hyb_2 do not change the adversary's advantage.

Proof: The only change we made is programming the random oracle to let the reduction foresee the challenged bits in the cut-and-choose protocol part. □

Claim 3. We claim that the changes made in Hyb_3 only increase the adversary's advantage by a negligible factor. In particular, it increases the advantage by the adventage of an adversary breaking the decisional Diffie-Hellman assumption in group $\mathbb{G}_q$.

Proof: Let $\mathcal{D}_3$ be a distinguisher between Hyb_2 and Hyb_3. We will construct a reduction algorithm $\mathcal{R}_3$ that breaks the decisional Diffie-Hellman assumption in group $\mathbb{G}_q$ using $\mathcal{D}_3$ as a sub procedure. Given an instance of the DDH problem (g_q^a, g_q^b, g_q^c) the reduction first uses the self-reducibility of this instance and computes for each $i \in J'$:

$$\begin{aligned}A_i &= (g_q^a)^{t_i}\\ B_i &= (g_q^b)^{t_i}\\ ct_i &\leftarrow \mathsf{PKE}^{\mathsf{nc}}.\mathsf{Enc}(ek, (g_q^c)^{t_i}; r_i)\end{aligned}$$

for some random $t_i \leftarrow\!\!\$\ \mathbb{Z}_q$. We also compute $Z_i := (g_q^c)^{t_i} \cdot v_i^{\mathsf{DVRF}}$. The rest of the values are computed as described by the protocol.It is easy to see if the provided tuple is a DDH tuple, then the reduction simulated Hyb_2, and otherwise it simulated Hyb_3 to the distinguisher $\mathcal{D}_3$. □

Claim 4. We claim that the experiment aborts because of the changes in Hyb_4 only with negligible probability. In particular, with advantage not bigger than the advantage of the decisional Diffie-Hellman assumption in group $\mathbb{G}_p$.

Proof: [Sketch.] We construct a reduction R_4 breaking the DDH assumption in $\mathbb{G}_p$ using an aborting adversary $\mathcal{D}_4$ as a sub procedure. The idea is that since $\mathcal{D}_4$ aborts before receiving the decryption key, dk cannot compute g_p^s (see Fig. Figure 5). Thus, we replace $(ek)^r$ with a random element from $\mathbb{G}_p$ for each ct_i, where $i \in J'$. We use a similar self-reducibility property of DDH tuples. If $\mathcal{D}_4$ distinguishes this change, then $\mathcal{R}_4$ breaks the DDH assumption; otherwise, $\mathcal{D}_4$ aborts by guessing g_p^s, which happens with negligible probability. □

We will now show how to construct simulators $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ from the simultability experiment in Hyb_4.

Simulator Sim_1:

This simulator gets as input the encryption key ek and the instance $inst$ and outputs the verifiable non-committing encryption ciphertext ct. For all opened sessions, i.e., for $j \in [2\lambda_s]/J'$, the simulator computes the values according to protocol. It is worth noting that since those sessions are part of $\mathcal{S}_{\mathsf{op}}$, it follows that the simulator does not have to compute the value Z_j, which includes requires the witness v_i^{DVRF} that is not given to Sim_1.

For the rest of the session, i.e., $j \in J'$, it computes A_j, B_j and ct_j as per the changes in hybrid Hyb_4, where $ct_j = (c_{1,j}, c_{2,j}, c_{3,j})$ and $c_{1,j} = g_p^{r_j}$, $c_{2,j} = (ek)^{r_j} \cdot g_p^{s_j}$ and $c_{3,j} \leftarrow\!\!\$\ \mathbb{G}_p$. Note that due to the change in Hyb_3, there exists a valid message for $c_{3,j}$ and due to the changes in Hyb_1, the proofs π_j are simulated using the ZK property, and no valid witness is required. The simulator outputs the state $\mathsf{st}_0' := ((j, c_j, g_p^{s_j}, c_{3,j})_{j \in J'})$.

It is worth noting that because of hybrid Hyb_2, the ciphertext ct will pass the verification VfEnc and, from the adversary's perspective, is a valid ciphertext.

Simulator Sim_2:

This simulator gets as input, the state st_0', the decryption key dk, the witness wit, auxilliary information z and outputs the decryption key dk. Having the witness v_i^{DVRF}, the simulator Sim_2 computes the values $C_j := v_i^{\mathsf{DVRF}} \cdot (Z_j)^{-1}$ for each $j \in J'$. The simulator now parses the state $st_0' = ((j, c_j, g_p^{s_j}, c_{3,j})_{j \in J'})$ and programs the random oracle H_p (see Fig. Figure 5) as follows. For each $j \in J'$ is sets $\mathsf{H}_p(g_p^{s_j}) := c_{3,j} \oplus C_j$. Finally, the simulator outputs the modified decryption key $dk' := dk$.

□

Appendix D. Full Proofs of Security

Proof: [Proof of Theorem 2] We now prove that our protocol in Figure 7 securely UC-realizes the functionality $\mathcal{F}_{\mathtt{swap}}$. We describe a simulator $\mathcal{S}_{\mathtt{swap}}$ that simulates the real-world execution protocol while interacting with the ideal functionality $\mathcal{F}_{\mathtt{swap}}$. We allow the PPT adversary to corrupt at most t servers and the client $\mathcal{C}$. The simulator will consider two cases separately. In the first case, we assume that the adversary corrupts the $\mathcal{C}$, while in the other one, it does not. We consider static corruption, in which the environment at the beginning of a session specifies the corrupted and honest parties. The simulator faithfully impersonates the honest party. The environment does not expect any interaction with the simulator for operations exclusively among corrupted users. Similarly, communications exclusively among honest nodes happen through secure channels. Therefore, the attacker does not gather additional information besides the fact that the communication took place, and we omit these operations in the description of our simulator. When describing the operations of simulator S, we initiate the discussion with a sequence of hybrids. These hybrids commence with a real-world execution and progressively modify the simulation in those hybrids. We then analyze the closeness between adjacent experiments. The simulator's execution for the functionality is defined as the final hybrid execution. Below, we outline the hybrid executions, followed by a discussion on their proximity. It is important to note that the transition between hybrid executions occurs per session, one at a time. We focus on a single instance in this discussion for simplicity and readability.

Hyb_0: This hybrid is the real world execution of the protocol in Figure 7.

Hyb_1: This is the same as the above execution except now, if in some session q_1 and for some honest server S_i, the adversary published signature $\sigma_{\mathsf{aux},i}$ on the blockchain for which $\mathsf{DS.Vf}(pk_{\mathsf{aux},i}, tx^j_{\mathsf{pay},i}, \sigma_{\mathsf{aux},i}) = 1$, where $\sigma_{\mathsf{aux},i}$ was not created by the simulator. We return abort_1 in such a case.

Hyb_2: This is the same as the above execution except now, if in some session q_2, for some honest server S_i and malicious client $\mathcal{C}$, the adversary sends an adaptor pre-signature for which

$$\mathsf{AS.pVf}(pk_{A,j}, tx^j_{\mathsf{pay},i}, \tilde{\sigma}_A, Y) = 1$$

but adaptation of the pre-signature to a signature fails. We return abort_2 in such a case.

Hyb_3: This is the same as the above execution except now, if in some session q_3 and honest $\mathcal{C}$, the adversary published signature $\sigma_{A,j}$ in the name of server S_j on the ledger/blockchain, but the extraction of the witness for the statement Y fails. We return abort_3 in such a case.

Hyb_4: Let $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ be the simulator from the simulatability property of the VNE scheme. This is the same as the above execution except now, for all honest servers, the simulator $\mathcal{S}_{\mathtt{swap}}$ uses $\mathsf{Sim}_1(ek, inst)$ to generate ct and later (after receiving a valid pre-signature) uses Sim_2 to compute the witness $y = dk$.

Hyb_5: This is the same as the above execution except now if in some session q_5 and corrupted $\mathcal{C}$, the adversary published a valid evaluation v under message m, together with a proof of evaluation π before any of the honest servers responds with its evaluation. We return abort_5 in such a case.

We will describe the inner workings of the simulator $\mathcal{S}_{\mathtt{swap}}$.

Key Generation. After receiving the set of corrupted servers C, the simulator aborts if $|\mathrm{C}| > t$. Otherwise, it sends $(\mathtt{keygen}, sid, \mathrm{C})$ to the functionality, initializing the session. It receives $(\mathtt{keygen}, sid, vk', \mathcal{I}, \{(sk'_i, vk'_i)\}_{i \in \mathrm{C}_\mathcal{I}})$ in return from it. The simulator also prepares receiving keys $\{rk_i\}_{i\in\mathrm{C}}$. Independently, the simulator executes the threshold key generation algorithm $(vk, (vk_j, sk_j)_{j\in[n]/[\mathrm{C}]}) \leftarrow \mathsf{DKgen}(1^\lambda, t, n)$ and sends the following information to the adversary $(vk, (vk_j, sk_j)_{j\in C})$. It also generates the transaction-related key pairs for all honest parties for the ledger/blockchain.

Honest Client. Upon receiving the message $(\mathtt{swap}_c, sid, vk, x, (t+1)d, pk_d, \mathcal{J})$ from the functionality, the simulator knows that the honest client started the protocol and expects the evaluation on point x. It, therefore, runs the setup phase of the protocol (i.e., generates and publishes the setup transaction). Once this setup is done, $\mathcal{S}_{\mathtt{swap}}$ awaits messages from the adversary, the functionality, and observes the ledger/blockchain.

The adversary initiates the real-world protocol with $\mathcal{S}_{\mathtt{swap}}$ in the name of corrupted server S_j. Both generate the payment transaction together and execute the sub-protocol $\Gamma^{\mathsf{2PC}}_{\mathsf{DVTS,DS}}$, exchanging the non-commiting VNE ciphertext and adaptor signatures. Once the adversary publishes $(tx^i_{\mathsf{pay},j}, \sigma_{A,i}, \sigma_{\mathsf{aux},k})$ on the ledger (for some i), the simulator uses the key sk'_j and queries the functionality with $(\mathtt{swap}_s, sid, vk', j, vk'_j, rk_j, y'_j)$, where $y'_j \leftarrow \mathtt{f.pareval}(x, sk'_j)$.

Upon receiving (sid, j) from the functionality, the simulator runs the real-world protocol in the name of the honest client and the honest Server S_j. It publishes $(tx^i_{\mathsf{pay},j}, \sigma_{A,i}, \sigma_{\mathsf{aux},j})$ on the ledger. It is worth noting that since the client is honest, the adversary is not receiving the evaluation for the honest servers and can only rely on the shares it received for malicious servers. Moreover, the above steps ensure that the ledger and the state of the functionality are consistent, i.e., the same parties received payment for their work.

Corrupted Client. In the first step, the simulator sends the message $(\mathtt{corrupt\text{-}client}, C)$ to the functionality, indicating that the client is corrupted. At some point, the adversary initiates the exchange by setting up all the transactions on the ledger/blockchain. The simulator prepares the deposit keypair (sk_d, pk_d) and uses the details from the adversary (including where point to be evaluated on x) and sends $(\mathtt{swap}_c, sid, vk, x, (t+1)d, sk_d, pk_d, \mathcal{J})$ to the functionality. Once the setup is done, $\mathcal{S}_{\mathtt{swap}}$ awaits messages from the adversary from the functionality and observes the ledger/blockchain.

The adversary initiates the real-world protocol with $\mathcal{S}_{\mathtt{swap}}$ in the name of the corrupted client and an honest server S_j. Both generate the payment transaction together and execute the sub-protocol $\Gamma^{\mathsf{2PC}}_{\mathsf{DVTS,DS}}$, exchanging the non-committing VNE ciphertext and adaptor signatures as in the previous hybrid. Once the functionality sends (sid, j) to the simulator, it signals that the real-world protocol needs to be finished. In such a case, the simulator publishes $(tx^i_{\mathsf{pay},j}, \sigma_{A,i}, \sigma_{\mathsf{aux},k})$ on the ledger (for some i).

At some point, the adversary outputs a valid evaluation v and proof of evaluation π for the distributed verifiable threshold service, the verification key for this session, and message x. In such a case, the simulator sends $\{\mathtt{obtain\_vrf}, sid, x\}$ to the functionality.

Claim 5. Hybrids Hyb_0 and Hyb_1 are indistinguishable, if DS strongly unforgeable under chosen message attacks.

Proof: If the event abort_1 occurs, then the adversary $\mathcal{A}$ can break the unforgeability of the DS scheme. We will show this by constructing a reduction algorithm $\mathcal{R}$ using the adversary to break the unforgeability of the DS scheme. The reduction randomly picks one of the honest server i indices and then sets its transaction public key to the key the reduction received from its DS challenger. If the honest server is required to sign during simulation, the reduction $\mathcal{R}$ uses its signing oracle to sign the transaction. At some point, the adversary $\mathcal{A}$ will publish a signature for server i that the reduction $\mathcal{R}$ did not generate (i.e., never asked for its signing oracle). This signature and transaction are a valid forgery that the reduction can now return and break the unforgeability of the DS scheme. □

Claim 6. Hybrids Hyb_1 and Hyb_2 are indistinguishable, if AS is pre-signature adaptable.

Proof: In case the event abort_2 occurs, then the adversary can be used to break the pre-signature adaptability of the AS scheme. □

Claim 7. Hybrids Hyb_2 and Hyb_3 are indistinguishable, if AS is witness extractable.

Proof: In case the event abort_3 occurs, then the adversary can be used to break the witness extractability property of the AS scheme. □

Claim 8. Hybrids Hyb_3 and Hyb_4 are indistinguishable, if VNE is simulatable.

Proof: This follows directly from the definition of simulatability of the verifiable non-committing encryption scheme VNE. □

Claim 9. Hybrids Hyb_4 and Hyb_5 are indistinguishable, if distributed verifiable threshold service is unpredictable.

Proof: The proof follows by a reduction to the unpredictability property. The idea behind the reduction is to simulate the protocol to the adversary while simultaneously playing the adversary's role in the unpredictability experiment. The reduction starts by defining the set of corrupted parties based on the adversary's choice and receiving the corresponding secret keys it can share with the adversary. For queries that require the reduction to use the DVTS primitive, it can query the $\mathcal{O}_{\mathsf{PartEval}}$ oracle. Finally, the adversary outputs (m, v, π) such that $\mathsf{Verify}(vk, (vk_j)_{j\in[n]}, m, v, \pi) = 1$ and we return the abort_5 event. The reduction can now return (m, v, π) as a forgery for the unpredictability property. Note that this is a valid forgery since it does not use the $\mathcal{O}_{\mathsf{PartEval}}$ for the message m. □

□

Appendix E. Proof of correctness of Z_j

The proof of correctness of $Z_j := A_j^{b_j} \cdot v_i^{\mathsf{DVRF}}$ is performed through a Schnorr proof. Here, we show how to prove the correctness of $g_1^{a_1} \cdot g_2^{a_2}$ where a_1 and a_2 are the scalar witnesses. Here, g_1, g_2 are the generators of the same cyclic group. The prover samples two random scalar values r_1, r_2 and for a random challenge c, computes a_1c+r_1 and a_2c+r_2 and forwards to the verifier. The challenge c can either be randomly chosen by the verifier or, for a non-interactive version, can be generated as an output of a hash function through the Fiat-Shamir heuristic. The prover also forwards the commitments $g_1^{r_1}$ and $g_2^{r_2}$ for the random values r_1, r_2. The verifier checks if $(g_1^{a_1} \cdot g_2^{a_2})^c \cdot g_1^{r_1} \cdot g_2^{r_2} = g_1^{a_1c+r_1} \cdot g_2^{a_2c+r_2}$. If yes, the proof is accepted.