UC Priced Oblivious Transfer with Purchase Statistics and Dynamic Pricing

Aditya Damodaran[1], Maria Dubovitskaya[2], and Alfredo Rial[1]

[1] SnT, University of Luxembourg
firstname.lastname@uni.lu
[2] Dfinity
maria@dfinity.org

Abstract. Priced oblivious transfer (POT) is a cryptographic protocol that can be used to protect customer privacy in e-commerce applications. Namely, it allows a buyer to purchase an item from a seller without disclosing to the latter which item was purchased and at which price. Unfortunately, existing POT schemes have some drawbacks in terms of design and functionality. First, the design of existing POT schemes is not modular. Typically, a POT scheme extends a k-out-of-N oblivious transfer (OT) scheme by adding prices to the items. However, all POT schemes do not use OT as a black-box building block with certain security guarantees. Consequently, security of the OT scheme needs to be reanalyzed while proving security of the POT scheme, and it is not possible to swap the underlying OT scheme with any other OT scheme. Second, existing POT schemes do not allow the seller to obtain any kind of statistics about the buyer's purchases, which hinders customer and sales management. Moreover, the seller is not able to change the prices of items without restarting the protocol from scratch.
We propose a POT scheme that addresses the aforementioned drawbacks. We prove the security of our POT in the UC framework. We modify a standard POT functionality to allow the seller to receive aggregate statistics about the buyer's purchases and to change prices dynamically. We present a modular construction for POT that realizes our functionality in the hybrid model. One of the building blocks is an ideal functionality for OT. Therefore, our protocol separates the tasks carried out by the underlying OT scheme from the additional tasks needed by a POT scheme. Thanks to that, our protocol is a good example of modular design and can be instantiated with any secure OT scheme as well as other building blocks without reanalyzing security from scratch.

Keywords: Oblivious transfer, UC security, modular design

1 Introduction

Priced oblivious transfer (POT) [1] is a cryptographic protocol that can be used to protect privacy in e-commerce applications. POT is a protocol between a seller or vendor $\mathcal{V}$ and a buyer $\mathcal{B}$. $\mathcal{V}$ sells N items (represented as messages) $\langle m_n \rangle_{n=1}^{N}$ with prices $\langle p_n \rangle_{n=1}^{N}$ assigned to them. At any transfer phase, $\mathcal{B}$ chooses an index $\sigma \in [1, N]$ and purchases the message m_σ. Security for $\mathcal{B}$ ensures that $\mathcal{V}$ does not learn the index σ or the price p_σ

paid by $\mathcal{B}$. Security for $\mathcal{V}$ ensures that $\mathcal{B}$ pays the correct price p_σ for the message m_σ and that $\mathcal{B}$ does not learn any information about the messages that are not purchased.

Typically, POT schemes use a prepaid mechanism [1,6,23,24,22,2]. $\mathcal{B}$ makes an initial deposit dep to $\mathcal{V}$, revealing the amount dep to $\mathcal{V}$. $\mathcal{B}$ and $\mathcal{V}$ can use an existing payment mechanism of their choice to carry out this transaction. After the deposit phase, when $\mathcal{B}$ purchases a message m_σ, the price p_σ is subtracted from the deposit, but the POT protocol ensures that: (1) $\mathcal{V}$ does not learn the new value $dep' = dep - p_\sigma$ of the deposit and (2) $dep' \geq 0$.

Lack of Modular Design. POT schemes [1,6,23,24,22,2] have so far been built by extending an existing oblivious transfer (OT) scheme. OT is a protocol between a sender $\mathcal{E}$ and a receiver $\mathcal{R}$. $\mathcal{E}$ inputs N messages $\langle m_n \rangle_{n=1}^{N}$. At each transfer phase, $\mathcal{R}$ obtains the message m_σ for her choice $\sigma \in [1, N]$. $\mathcal{E}$ does not learn σ, while $\mathcal{R}$ does not learn any information about other messages.

In OT schemes that have been used to build POT schemes, the interaction between $\mathcal{E}$ and $\mathcal{R}$ consists of an initialization phase followed by several transfer phases. In the initialization phase, $\mathcal{E}$ encrypts messages $\langle m_n \rangle_{n=1}^{N}$ and sends the list of ciphertexts to $\mathcal{R}$. In each transfer phase, $\mathcal{R}$, on input σ, computes a blinded request for $\mathcal{E}$. $\mathcal{E}$ sends a response that allows $\mathcal{R}$ to decrypt the ciphertext that encrypts m_σ.

Roughly speaking, to construct a POT scheme from an OT scheme, typically the OT scheme is extended as follows. First, in the initialization phase, the computation of the ciphertexts is modified in order to bind them to the prices of the encrypted messages, e.g. by using a signature scheme. Second, a deposit phase, where $\mathcal{B}$ sends an initial deposit to $\mathcal{V}$, is added. As a result of this deposit phase, $\mathcal{V}$ and $\mathcal{B}$ output a commitment or encryption to the deposit dep. Third, in each transfer phase, the request computed in the OT scheme is extended by $\mathcal{B}$ in order to send to $\mathcal{V}$ an encryption or commitment to the new value dep' of the deposit and to prove to $\mathcal{V}$ (e.g. by using a zero-knowledge proof) that $dep' = dep - p_\sigma$ and that $dep' \geq 0$.

The main drawback of the design of existing POT schemes is a lack of modularity. Although each POT scheme is based on an underlying OT scheme, the latter is not used as a black-box building block. Instead, every OT scheme is modified and extended ad-hoc to create the POT scheme, blurring what components were present in the original OT scheme and what components were added to create the POT scheme.

The lack of modularity has two disadvantages. First, existing POT schemes cannot easily be modified to use another OT scheme as a building block, for example, a more efficient one. Second, every time a new POT scheme is designed, the proofs need to be done from scratch. This means that the security of the underlying OT scheme will be reanalyzed instead of relying on its security guarantees. This is error-prone.

Lack of Purchase Statistics. POT schemes [1,6,23,24,22,2] effectively prevent $\mathcal{V}$ from learning what messages are purchased by $\mathcal{B}$. Although this is a nice privacy feature for $\mathcal{B}$, the customer and sales management becomes more difficult for $\mathcal{V}$. For example, $\mathcal{V}$ is not able to know which items are more demanded by buyers and which ones sell poorly. As another example, $\mathcal{V}$ is not able to use marketing techniques like giving discounts that depend on the previous purchases of a buyer. It is desirable that, while protecting privacy of each individual purchase, $\mathcal{V}$ gets some aggregate statistics about $\mathcal{B}$'s purchases.

Lack of Dynamic Pricing. In existing POT schemes [1,6,23,24,22,2], the price of a message is static, i.e. each message is associated with a price in the initialization phase and that price cannot change afterwards. In practical e-commerce settings, this is undesirable because sellers would like to be able to change the price of a product easily. However, modifying existing POT schemes to allow sellers to change the prices of messages at any time throughout the protocol execution is not straightforward and would require rerunning the initialization phase.

1.1 Our Contribution

Functionality $\mathcal{F}_{\mathrm{POTS}}$. We use the universal composability (UC) framework [12] and we describe an ideal functionality $\mathcal{F}_{\mathrm{POTS}}$ for priced oblivious transfer with purchase statistics and dynamic pricing. We modify a standard POT functionality to enable aggregate statistics and dynamic pricing.

Existing functionalities for POT [6,23] consist of three interfaces: an initialization interface where $\mathcal{V}$ sends the messages $\langle m_n \rangle_{n=1}^{N}$ and the prices $\langle p_n \rangle_{n=1}^{N}$ to the functionality; a deposit interface where $\mathcal{B}$ sends a deposit dep to the functionality, which reveals dep to $\mathcal{V}$; and a transfer interface where $\mathcal{B}$ sends an index $\sigma \in [1, N]$ to the functionality and receives the message m_σ from the functionality if the current deposit is higher than the price of the message. The functionality stores the updated value of the deposit.

Our functionality $\mathcal{F}_{\mathrm{POTS}}$ modifies the initialization interface so that $\mathcal{V}$ only inputs the messages $\langle m_n \rangle_{n=1}^{N}$. Additionally, this interface can be invoked multiple times to send different tuples $\langle m_n \rangle_{n=1}^{N}$ of messages, where each tuple is associated with a unique epoch identifier ep. The idea is that messages of different epochs but with the same message index correspond to the same type or category of items. (This happens, e.g. when using POT to construct a conditional access system for pay-TV [2].) $\mathcal{F}_{\mathrm{POTS}}$ also modifies the transfer interface in order to store the number of times that $\mathcal{B}$ purchases items of each of the types or categories.

Moreover, $\mathcal{F}_{\mathrm{POTS}}$ adds three new interfaces: a "setup price" interface where $\mathcal{V}$ inputs the prices, an "update price" interface where $\mathcal{V}$ modifies the price of a message, and a "reveal statistic" interface where $\mathcal{F}_{\mathrm{POTS}}$ reveals to $\mathcal{V}$ the value of a statistic about the purchases of $\mathcal{B}$.

We propose a scheme Π_{POTS} that realizes $\mathcal{F}_{\mathrm{POTS}}$. Π_{POTS} is designed modularly and provides purchase statistics and dynamic pricing, as described below.

Modular Design. In the UC framework, protocols can be described modularly by using a hybrid model where parties invoke the ideal functionalities of the building blocks of a protocol. For example, consider a protocol that uses as building blocks a zero-knowledge proof of knowledge and a signature scheme. In a modular description of this protocol in the hybrid model, parties in the real world invoke the ideal functionalities for zero-knowledge proofs and for signatures.

We describe Π_{POTS} modularly in the hybrid model. Therefore, $\mathcal{V}$ and $\mathcal{B}$ in the real world invoke only ideal functionalities for the building blocks of Π_{POTS}. Interestingly, one of the building blocks used in Π_{POTS} is the ideal functionality for oblivious transfer $\mathcal{F}_{\mathrm{OT}}$. Thanks to that, Π_{POTS} separates the task that is carried out by the underlying OT scheme from the additional tasks that are needed to create a POT scheme.

The advantages of a modular design are twofold. First, $\mathit{\Pi}_{\mathrm{POTS}}$ can be instantiated with any secure OT scheme, i.e., any scheme that realizes $\mathcal{F}_{\mathrm{OT}}$. The remaining building blocks can also be instantiated with any scheme that realizes their corresponding ideal functionalities, leading to multiple possible instantiations of $\mathit{\Pi}_{\mathrm{POTS}}$. Second, the security analysis in the hybrid model is simpler and does not need to reanalyze the security of any of the building blocks.

One challenge when describing a UC protocol in the hybrid model is the need to ensure that two or more ideal functionalities receive the same input. For example, in $\mathit{\Pi}_{\mathrm{POTS}}$, it is necessary to enforce that $\mathcal{B}$ sends the same index $\sigma \in [1, N]$ to the transfer interface of $\mathcal{F}_{\mathrm{OT}}$ and to another functionality $\mathcal{F}_{\mathrm{NHCD}}$ (described below) that binds σ to the price p_σ. Otherwise, if an adversarial buyer sends different indexes σ and σ' to $\mathcal{F}_{\mathrm{OT}}$ and $\mathcal{F}_{\mathrm{NHCD}}$, $\mathcal{B}$ could obtain the message m_σ and pay an incorrect price $p_{\sigma'}$. To address this issue, we use the method proposed in [9], which uses a functionality $\mathcal{F}_{\mathrm{NIC}}$ for non-interactive commitments.

Purchase Statistics. In $\mathit{\Pi}_{\mathrm{POTS}}$, $\mathcal{V}$ can input multiple tuples $\langle m_n \rangle_{n=1}^{N}$ of messages. We consider that messages associated with the same index σ belong to the same category.

$\mathit{\Pi}_{\mathrm{POTS}}$ allows $\mathcal{B}$ to reveal to $\mathcal{V}$ information related to how many purchases were made for each of the item categories. To do that, $\mathcal{B}$ stores a table Tbl_{st} of counters of how many purchases were made for each category. Tbl_{st} contains position-value entries $[\sigma, v_\sigma]$, where $\sigma \in [1, N]$ is the category index and v_σ is the counter. Any time a message m_σ is purchased, the counter for category σ is incremented in Tbl_{st}. At any time throughout the execution of $\mathit{\Pi}_{\mathrm{POTS}}$, $\mathcal{B}$ can choose a statistic ST, evaluate it on input Tbl_{st} and reveal to $\mathcal{V}$ the result.

Additionally, $\mathcal{B}$ must prove to $\mathcal{V}$ that the result is correct. To do that, we need a mechanism that allows $\mathcal{V}$ to keep track of the purchases made $\mathcal{B}$, without learning them. For this purpose, $\mathit{\Pi}_{\mathrm{POTS}}$ uses the functionality for a committed database $\mathcal{F}_{\mathrm{CD}}$ recently proposed in [8]. $\mathcal{F}_{\mathrm{CD}}$ stores the table Tbl_{st} of counters and allows $\mathcal{B}$ to read the counters from Tbl_{st} and to write updated counters into Tbl_{st} each time a purchase is made. $\mathcal{V}$ does not learn any information read or written but is guaranteed of the correctness of that information. $\mathcal{F}_{\mathrm{CD}}$ allows $\mathcal{B}$ to hide from $\mathcal{V}$ not only the value of counters read or written, but also the positions where they are read or written into Tbl_{st}. This is a crucial property to construct $\mathit{\Pi}_{\mathrm{POTS}}$, because the position read or written is equal to the index σ, which needs to be hidden from $\mathcal{V}$ in order to hide what message is purchased. The method in [9] is used to ensure that the index σ is the same both for the counter v_σ incremented in $\mathcal{F}_{\mathrm{CD}}$ and for the message m_σ obtained through $\mathcal{F}_{\mathrm{OT}}$. In [8], an efficient construction for $\mathcal{F}_{\mathrm{CD}}$ based on vector commitments (VC) [20,15] is provided, where a VC commits to a vector x such that $x[\sigma] = v_\sigma$ for $\sigma \in [1, N]$. In this construction, after setup, the efficiency of the read and write operations does not depend on the size of the table, which yields efficient instantiations of $\mathit{\Pi}_{\mathrm{POTS}}$ when N is large.

We note that $\mathcal{V}$ could be more interested in gathering aggregated statistics about multiple buyers rather than a single buyer. Interestingly, $\mathit{\Pi}_{\mathrm{POTS}}$ opens up that possibility. The functionalities $\mathcal{F}_{\mathrm{CD}}$ used between $\mathcal{V}$ and each of the buyers can be used in a secure multiparty computation (MPC) protocol for the required statistic. In this MPC, each buyer uses $\mathcal{F}_{\mathrm{CD}}$ to read and prove correctness of the information about her purchases. With the instantiation of $\mathcal{F}_{\mathrm{CD}}$ based on a VC scheme, $\mathcal{V}$ and the buyers would run a secure MPC protocol where $\mathcal{V}$ inputs one vector commitment for each buyer and each buyer inputs the committed vector and the opening.

Dynamic Pricing. In existing POT schemes [1,6,23,24,22,2], when $\mathcal{V}$ encrypts the messages $\langle m_n \rangle_{n=1}^{N}$ in the initialization phase, the price of the encrypted message is somehow bound to the corresponding ciphertext. This binding is done in such a way that, when $\mathcal{B}$ computes a request to purchase a message m_σ, $\mathcal{V}$ is guaranteed that the correct price p_σ is subtracted from the deposit while still not learning p_σ. In some schemes, $\mathcal{V}$ uses a signature scheme to sign the prices in the initialization phase, and $\mathcal{B}$ uses a zero-knowledge proof of signature possession to compute the request.

It would be possible to modify the initialization phase of those schemes so that ciphertexts on the messages $\langle m_n \rangle_{n=1}^{N}$ are computed independently of the signatures on the prices $\langle p_n \rangle_{n=1}^{N}$, yet enforcing that requests computed by $\mathcal{B}$ use the right price p_σ for the requested index σ. (For example, both the ciphertext and the signature could embed the index σ, and $\mathcal{B}$, as part of a request, would be required to prove in zero-knowledge that the index in the ciphertext and in the signature are equal.) This would allow $\mathcal{V}$ to modify the prices of messages by issuing new signatures, without needing to re-encrypt the messages. However, this mechanism to update prices would also require some method to revoke the previous signatures, which would heavily affect the efficiency to the Π_{POTS} protocol.

Instead, we use the functionality $\mathcal{F}_{\mathrm{NHCD}}$ for a non-hiding committed database recently proposed in [21]. $\mathcal{F}_{\mathrm{NHCD}}$ stores a table Tbl_{nhcd} with entries $[\sigma, p_\sigma]$, where $\sigma \in [1, N]$ is an index and p_σ is a price. $\mathcal{V}$ sets the initial values of Tbl_{nhcd} and is also able to modify Tbl_{nhcd} at any time. $\mathcal{B}$ knows the content of Tbl_{nhcd} but cannot modify it. When purchasing a message of index σ, $\mathcal{B}$ reads from $\mathcal{F}_{\mathrm{NHCD}}$ the entry $[\sigma, p_\sigma]$. $\mathcal{V}$ does not learn any information about the entry read, yet $\mathcal{V}$ is guaranteed that a valid entry is read. Similarly to the case of $\mathcal{F}_{\mathrm{CD}}$, we stress that $\mathcal{F}_{\mathrm{NHCD}}$ reveals to $\mathcal{V}$ neither the position σ nor the value p_σ, and that the method in [9] is used to prove that the index σ received by $\mathcal{F}_{\mathrm{NHCD}}$ and by $\mathcal{F}_{\mathrm{OT}}$ are the same. In [21], an efficient construction for $\mathcal{F}_{\mathrm{NHCD}}$ based on a non-hiding VC scheme [20,15] is provided, where a non-hiding VC commits to a vector x such that $x[\sigma] = p_\sigma$ for $\sigma \in [1, N]$. In this construction, after setup, the efficiency of the read and write operations does not depend on the size of the table, which yields efficient instantiations of Π_{POTS} when N is large.

It could seem that dynamic pricing undermines buyer's privacy in comparison to existing POT schemes, e.g. when an adversarial $\mathcal{V}$ offers different prices to each of the buyers and changes them dynamically in order to narrow down the messages that a buyer could purchase. However, this is not the case. In existing POT, a seller is also able to offer different prices to each buyer, and to change them dynamically by restarting the protocol. The countermeasure, for both Π_{POTS} and other POT, is to have lists of prices available through a secure bulletin board where buyers can check that the prices they are offered are equal to those for other buyers.

2 Related Work

POT was initially proposed in [1]. The POT scheme in [1] is secure in a half-simulation model, where a simulation-based security definition is used to protect seller security, while an indistinguishability-based security definition is used to protect buyer privacy. Later, POT schemes in the full-simulation model [6] and UC-secure schemes [23] were proposed. The scheme in [6] provides unlinkability between $\mathcal{V}$ and $\mathcal{B}$, i.e., $\mathcal{V}$ cannot link interactions with the same buyer.

We define security for POT in the UC model, like [23], and our protocol does not provide unlinkability, unlike [6] or the PIR-based scheme in [18]. Although unlinkability is important in some settings, an unlinkable POT scheme would require the use of an anonymous communication network and, in the deposit phase, it would hinder the use of widespread payment mechanisms that require authentication. Therefore, because one of the goals of this work is to facilitate sellers' deployment of POT schemes, we chose to describe a scheme that does not provide unlinkability.

The use of POT as building block in e-commerce applications in order to protect buyer privacy has been described, e.g. in the context of buyer-seller watermarking protocols for copyright protection [22] and conditional access systems for pay-TV [2]. Our POT protocol is suitable to be used in any of the proposed settings and it provides additional functionalities to $\mathcal{V}$. In [24], a transformation that takes a POT scheme and produces a POT scheme with optimistic fair exchange is proposed. This transformation can also be used with our POT scheme.

Oblivious transfer with access control (OTAC) [16,5] is a generalization of oblivious transfer where messages are associated with access control policies. In order to obtain a message, a receiver must prove that she fulfils the requirements described in the associated access control policy. In some schemes, access control policies are public [16,5,25,19], while other schemes hide them from the receiver [7,4].

POT could be seen as a particular case of OTAC with public access control policies. In POT, the public access control policy that $\mathcal{B}$ must fulfil to get a message is defined as her current deposit being higher than the price of the message. However, existing OTAC schemes cannot straightforwardly be converted into a POT scheme. The reason is that, in adaptive POT, the fulfilment of a policy by $\mathcal{B}$ depends on the history of purchases and deposits of $\mathcal{B}$, i.e., whether or not the current deposit of $\mathcal{B}$ allows him to buy a message depends on how much $\mathcal{B}$ deposited and spent before. Therefore, POT schemes need to implement a mechanism that allows $\mathcal{V}$ to keep track of the current deposit of $\mathcal{B}$ without learning it, such as a commitment or an encryption of the deposit that is updated by $\mathcal{B}$ at each deposit or purchase phase. (Our POT protocol uses functionality $\mathcal{F}_{\mathrm{CD}}$ to store the deposit, in addition to the counters of purchases.) In contrast, existing OTAC schemes do not provide such a mechanism. In those schemes, usually a third party called issuer certifies the attributes of a receiver, and after that the receiver can use those certifications to prove the she fulfils an access control policy.

The oblivious language-based envelope (OLBE) framework in [3] generalizes POT, OTAC and similar protocols like conditional OT. However, similar to the case of OTAC schemes, the instantiation of POT in the OLBE framework is only straightforward for non-adaptive POT schemes, where $\mathcal{V}$ does not need to keep track of the deposit of $\mathcal{B}$.

Aside from solutions based on OT, privacy protection in e-commerce can also be provided by protocols that offer anonymity/unlinkability. Here the goal is to protect the identity of $\mathcal{B}$ rather than the identity of the items purchased. Most solutions involve anonymous payment methods [10] and anonymous communication networks [17].

3 Universally Composable Security

We prove our protocol secure in the universal composability framework [12]. The UC framework allows one to define and analyze the security of cryptographic protocols so that security is retained under an arbitrary composition with other protocols. The security of a protocol is defined by means of an ideal protocol that carries out the desired task. In the ideal protocol, all parties send their inputs to an ideal functionality $\mathcal{F}$ for the task. $\mathcal{F}$ locally computes the outputs of the parties and provides each party with its prescribed output.

The security of a protocol φ is analyzed by comparing the view of an environment $\mathcal{Z}$ in a real execution of φ against that of $\mathcal{Z}$ in the ideal protocol defined in $\mathcal{F}_{\varphi}$. $\mathcal{Z}$ chooses the inputs of the parties and collects their outputs. In the real world, $\mathcal{Z}$ can communicate freely with an adversary $\mathcal{A}$ who controls both the network and any corrupt parties. In the ideal world, $\mathcal{Z}$ interacts with dummy parties, who simply relay inputs and outputs between $\mathcal{Z}$ and $\mathcal{F}_{\varphi}$, and a simulator $\mathcal{S}$. We say that a protocol φ securely realizes $\mathcal{F}_{\varphi}$ if $\mathcal{Z}$ cannot distinguish the real world from the ideal world, i.e., $\mathcal{Z}$ cannot distinguish whether it is interacting with $\mathcal{A}$ and parties running protocol φ or with $\mathcal{S}$ and dummy parties relaying to $\mathcal{F}_{\varphi}$.

A protocol $\varphi^{\mathcal{G}}$ securely realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model when φ is allowed to invoke the ideal functionality $\mathcal{G}$. Therefore, for any protocol ψ that securely realizes $\mathcal{G}$, the composed protocol φ^{ψ}, which is obtained by replacing each invocation of an instance of $\mathcal{G}$ with an invocation of an instance of ψ, securely realizes $\mathcal{F}$.

In the ideal functionalities described in this paper, we consider static corruptions. When describing ideal functionalities, we use the following conventions as in [9].

Interface Naming Convention. An ideal functionality can be invoked by using one or more interfaces. The name of a message in an interface consists of three fields separated by dots, e.g., pot.init.ini in the priced oblivious transfer functionality described in Section 4. The first field indicates the name of the functionality and is the same in all interfaces of the functionality. This field is useful for distinguishing between invocations of different functionalities in a hybrid protocol that uses two or more different functionalities. The second field indicates the kind of action performed by the functionality and is the same in all messages that the functionality exchanges within the same interface. The third field distinguishes between the messages that belong to the same interface, and can take the following different values. A message pot.init.ini is the incoming message received by the functionality, i.e., the message through which the interface is invoked. A message pot.init.end is the outgoing message sent by the functionality, i.e., the message that ends the execution of the interface. The message pot.init.sim is used by the functionality to send a message to $\mathcal{S}$, and the message pot.init.rep is used to receive a message from $\mathcal{S}$. The message pot.init.req is used by the functionality to send a message to $\mathcal{S}$ to request the description of algorithms from $\mathcal{S}$, and the message pot.init.alg is used by $\mathcal{S}$ to send the description of those algorithms to the functionality.

Network vs local communication. The identity of an interactive Turing machine instance (ITI) consists of a party identifier pid and a session identifier sid. A set of parties in an execution of a system of interactive Turing machines is a protocol instance if they have the same session identifier sid. ITIs can pass direct inputs to and outputs from "local" ITIs that have the same pid. An ideal functionality $\mathcal{F}$ has $pid = \perp$ and is considered local to all parties. An instance of $\mathcal{F}$ with the session identifier sid only accepts inputs from and passes outputs to machines with the same session identifier sid. Some functionalities require the session identifier to have some structure. Those functionalities check whether the session identifier possesses the required structure in the first message that invokes the functionality. For the subsequent messages, the functionality implicitly checks that the session identifier equals the session identifier used in the first message. Communication between ITIs with different party identifiers must take place over the network. The network is controlled by $\mathcal{A}$, meaning that he can arbitrarily delay, modify, drop, or insert messages.

Query identifiers. Some interfaces in a functionality can be invoked more than once. When the functionality sends a message $\mathsf{pot.init.sim}$ to $\mathcal{S}$ in such an interface, a query identifier qid is included in the message. The query identifier must also be included in the response $\mathsf{pot.init.rep}$ sent by $\mathcal{S}$. The query identifier is used to identify the message $\mathsf{pot.init.sim}$ to which $\mathcal{S}$ replies with a message $\mathsf{pot.init.rep}$. We note that, typically, $\mathcal{S}$ in the security proof may not be able to provide an immediate answer to the functionality after receiving a message $\mathsf{pot.init.sim}$. The reason is that $\mathcal{S}$ typically needs to interact with the copy of $\mathcal{A}$ it runs in order to produce the message $\mathsf{pot.init.rep}$, but $\mathcal{A}$ may not provide the desired answer or may provide a delayed answer. In such cases, when the functionality sends more than one message $\mathsf{pot.init.sim}$ to $\mathcal{S}$, $\mathcal{S}$ may provide delayed replies, and the order of those replies may not follow the order of the messages received.

Aborts. When an ideal functionality $\mathcal{F}$ aborts after being activated with a message sent by a party, we mean that $\mathcal{F}$ halts the execution of its program and sends a special abortion message to the party that invoked the functionality. When an ideal functionality $\mathcal{F}$ aborts after being activated with a message sent by $\mathcal{S}$, we mean that $\mathcal{F}$ halts the execution of its program and sends a special abortion message to the party that receives the outgoing message from $\mathcal{F}$ after $\mathcal{F}$ is activated by $\mathcal{S}$.

Delayed outputs. We say that an ideal functionality $\mathcal{F}$ *sends a public delayed output* v to a party $\mathcal{P}$ if it engages in the following interaction. $\mathcal{F}$ sends to $\mathcal{S}$ a note that it is ready to generate an output to $\mathcal{P}$. The note includes the value v, the identity $\mathcal{P}$, and a unique identifier for this output. When $\mathcal{S}$ replies to the note by echoing the unique identifier, $\mathcal{F}$ outputs the value v to $\mathcal{P}$. A *private delayed output* is similar, but the value v is not included in the note.

4 Ideal Functionality for POT with Statistics and Dynamic Pricing

We depict our functionality $\mathcal{F}_{\mathrm{POTS}}$ for POT with purchase statistics and dynamic pricing. $\mathcal{F}_{\mathrm{POTS}}$ interacts with a seller $\mathcal{V}$ and with a buyer $\mathcal{B}$ and consists of the following interfaces:

1. $\mathcal{V}$ uses the $\mathsf{pot.init}$ interface to send a list of messages $\langle m_n \rangle_{n=1}^{N}$ and an epoch identifier ep to $\mathcal{F}_{\mathrm{POTS}}$. $\mathcal{F}_{\mathrm{POTS}}$ stores $\langle m_n \rangle_{n=1}^{N}$ and ep, and sends N and ep to $\mathcal{B}$. In the first invocation of this interface, $\mathcal{F}_{\mathrm{POTS}}$ also initializes a deposit dep' and a

table Tbl_{st} with entries of the form $[\sigma, v_\sigma]$, where $\sigma \in [1, \mathcal{N}_{max}]$ is a category and v_σ is a counter of the number of purchases made for that category.

2. $\mathcal{V}$ uses the pot.setupprices interface to send a list of prices $\langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}$ to $\mathcal{F}_{\mathrm{POTS}}$, where $\mathcal{N}_{max}$ is the maximum number of messages in an epoch. $\mathcal{F}_{\mathrm{POTS}}$ stores $\langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}$ and sends $\langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}$ to $\mathcal{B}$.
3. $\mathcal{V}$ uses the pot.updateprice interface to send an index n and a price p to $\mathcal{F}_{\mathrm{POTS}}$. $\mathcal{F}_{\mathrm{POTS}}$ updates the stored list $\langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}$ with p at position n, and sends n and p to $\mathcal{B}$.
4. $\mathcal{B}$ uses the pot.deposit interface to send a deposit dep to $\mathcal{F}_{\mathrm{POTS}}$. $\mathcal{F}_{\mathrm{POTS}}$ updates the stored deposit $dep' \leftarrow dep' + dep$ and sends dep to $\mathcal{V}$.
5. $\mathcal{B}$ uses the pot.transfer interface to send an epoch ep and an index σ to $\mathcal{F}_{\mathrm{POTS}}$. If $dep' \geq p_\sigma$, $\mathcal{F}_{\mathrm{POTS}}$ increments the counter for category σ in Tbl_{st} and sends m_σ for the epoch ep to $\mathcal{B}$.
6. $\mathcal{B}$ uses the pot.revealstatistic interface to send a function ST to $\mathcal{F}_{\mathrm{POTS}}$. $\mathcal{F}_{\mathrm{POTS}}$ evaluates ST on input table Tbl_{st} and sends the result v and ST to $\mathcal{V}$. ST may be any function from a universe Ψ.

In previous functionalities for POT [6,23], $\mathcal{V}$ sends the messages and prices through the pot.init interface. In contrast, $\mathcal{F}_{\mathrm{POTS}}$ uses the pot.setupprices and pot.updateprice interfaces to send and update the prices. This change allows the design of a protocol where $\mathcal{V}$ can update prices without rerunning the initialization phase. We also note that, in $\mathcal{F}_{\mathrm{POTS}}$, all the messages m_σ of the same category σ have the same price for any epoch. The idea here is that messages of the same category represent the same type of content, which is updated by $\mathcal{V}$ at each new epoch. Nevertheless, it is straightforward to modify $\mathcal{F}_{\mathrm{POTS}}$ so that $\mathcal{V}$ can send a new list of prices for each epoch. Our construction in Section 6 can easily be modified to allow different prices for each epoch.

$\mathcal{F}_{\mathrm{POTS}}$ initializes a counter ct_v and a counter ct_b in the pot.setupprices interface. ct_v is incremented each time $\mathcal{V}$ sends the update of a price, and ct_b is incremented each time $\mathcal{B}$ receives the update of a price. These counters are used by $\mathcal{F}_{\mathrm{POTS}}$ to check that $\mathcal{V}$ and $\mathcal{B}$ have the same list of prices. We note that the simulator $\mathcal{S}$, when queried by $\mathcal{F}_{\mathrm{POTS}}$, may not reply or may provide a delayed response, which could prevent price updates sent by $\mathcal{V}$ to be received by $\mathcal{B}$.

The session identifier sid has the structure $(\mathcal{V}, \mathcal{B}, sid')$. This allows any vendor $\mathcal{V}$ to create an instance of $\mathcal{F}_{\mathrm{POTS}}$ with any buyer $\mathcal{B}$. After the first invocation of $\mathcal{F}_{\mathrm{POTS}}$, $\mathcal{F}_{\mathrm{POTS}}$ implicitly checks that the session identifier in a message is equal to the one received in the first invocation.

When invoked by $\mathcal{V}$ or $\mathcal{B}$, $\mathcal{F}_{\mathrm{POTS}}$ first checks the correctness of the input. Concretely, $\mathcal{F}_{\mathrm{POTS}}$ aborts if that input does not belong to the correct domain. $\mathcal{F}_{\mathrm{POTS}}$ also aborts if an interface is invoked at an incorrect moment in the protocol. For example, $\mathcal{V}$ cannot invoke pot.updateprice before pot.setupprices. Similar abortion conditions are listed when $\mathcal{F}_{\mathrm{POTS}}$ receives a message from the simulator $\mathcal{S}$.

Before $\mathcal{F}_{\mathrm{POTS}}$ queries $\mathcal{S}$, $\mathcal{F}_{\mathrm{POTS}}$ saves its state, which is recovered when receiving a response from $\mathcal{S}$. When an interface, e.g. pot.updateprice, can be invoked more than once, $\mathcal{F}_{\mathrm{POTS}}$ creates a query identifier qid, which allows $\mathcal{F}_{\mathrm{POTS}}$ to match a query to $\mathcal{S}$ to a response from $\mathcal{S}$. Creating qid is not necessary if an interface, such as pot.setupprices, can be invoked only once, or if it can be invoked only once with a concrete input revealed to $\mathcal{S}$, such as pot.init, which is invoked only once per epoch.

Compared to previous functionalities for POT, $\mathcal{F}_{\mathrm{POTS}}$ looks more complex. The reason is that we list all the conditions for abortion and that $\mathcal{F}_{\mathrm{POTS}}$ saves state information before querying $\mathcal{S}$ and recovers it after receiving a response from $\mathcal{S}$. These operations are also required but have frequently been omitted in the description of ideal functionalities in the literature. We describe $\mathcal{F}_{\mathrm{POTS}}$ below.

Description of $\mathcal{F}_{\mathrm{POTS}}$. Functionality $\mathcal{F}_{\mathrm{POTS}}$ runs with a seller $\mathcal{V}$ and a buyer $\mathcal{B}$, and is parameterised with a maximum number of messages $\mathcal{N}_{max}$, a message space $\mathcal{M}$, a maximum deposit value dep_{max}, a maximum price $\mathcal{P}_{max}$, and a universe of statistics Ψ that consists of ppt algorithms.

1. On input $(\mathsf{pot.init.ini}, sid, ep, \langle m_n \rangle_{n=1}^{N})$ from $\mathcal{V}$:
   - Abort if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$.
   - Abort if $(sid, ep', \langle m_n \rangle_{n=1}^{N}, 0)$, where $ep' = ep$, is already stored.
   - Abort if $N > \mathcal{N}_{max}$, or if for $n = 1$ to N, $m_n \notin \mathcal{M}$.
   - Store $(sid, ep, \langle m_n \rangle_{n=1}^{N}, 0)$.
   - Send $(\mathsf{pot.init.sim}, sid, ep, N)$ to $\mathcal{S}$.

S. On input $(\mathsf{pot.init.rep}, sid, ep)$ from $\mathcal{S}$:
   - Abort if $(sid, ep, \langle m_n \rangle_{n=1}^{N}, 0)$ is not stored, or if $(sid, ep, \langle m_n \rangle_{n=1}^{N}, 1)$ is already stored.
   - If a tuple (sid, Tbl_{st}) is not stored, initialize $dep' \leftarrow 0$ and a table Tbl_{st} with entries $[i, 0]$ for $i = 1$ to $\mathcal{N}_{max}$, and store $(sid, dep', \mathsf{Tbl}_{st})$.
   - Store $(sid, ep, \langle m_n \rangle_{n=1}^{N}, 1)$.
   - Send $(\mathsf{pot.init.end}, sid, ep, N)$ to $\mathcal{B}$.

2. On input $(\mathsf{pot.setupprices.ini}, sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$ from $\mathcal{V}$:
   - Abort if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$ or if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_v)$ is already stored.
   - Abort if, for $n = 1$ to $\mathcal{N}_{max}$, $p_n \notin (0, \mathcal{P}_{max}]$.
   - Initialize a counter $ct_v \leftarrow 0$ and store $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_v)$.
   - Send $(\mathsf{pot.setupprices.sim}, sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$ to $\mathcal{S}$.

S. On input $(\mathsf{pot.setupprices.rep}, sid)$ from $\mathcal{S}$:
   - Abort if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_v)$ is not stored, or if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_b)$ is already stored.
   - Initialize a counter $ct_b \leftarrow 0$ and store $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_b)$.
   - Send $(\mathsf{pot.setupprices.end}, sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$ to $\mathcal{B}$.

3. On input $(\mathsf{pot.updateprice.ini}, sid, n, p)$ from $\mathcal{V}$:
   - Abort if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_v)$ is not stored.
   - Abort if $n \notin [1, \mathcal{N}_{max}]$, or if $p \notin (0, \mathcal{P}_{max}]$.
   - Increment ct_v, set $p_n \leftarrow p$ and store them into the tuple $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_v)$.
   - Create a fresh qid and store (qid, n, p, ct_v).
   - Send $(\mathsf{pot.updateprice.sim}, sid, qid, n, p)$ to $\mathcal{S}$.

S. On input $(\mathsf{pot.updateprice.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if (qid, n, p, ct_v) is not stored, or if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_b)$ is not stored, or if $ct_v \neq ct_b + 1$.
   - Increment ct_b, set $p_n \leftarrow p$, and store them into the tuple $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_b)$.
   - Delete the record (qid, n, p, ct_v).

– Send $(\mathsf{pot.updateprice.end}, sid, n, p)$ to $\mathcal{B}$.

4. On input $(\mathsf{pot.deposit.ini}, sid, dep)$ from $\mathcal{B}$:
   – Abort if $(sid, dep', \mathsf{Tbl}_{st})$ is not stored, or if $dep' + dep \notin [0, dep_{max}]$.
   – Create a fresh qid and store (qid, dep).
   – Send $(\mathsf{pot.deposit.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{pot.deposit.rep}, sid, qid)$ from $\mathcal{S}$:
   – Abort if (qid, dep) is not stored.
   – Set $dep' \leftarrow dep' + dep$ and update $(sid, dep', \mathsf{Tbl}_{st})$.
   – Delete the record (qid, dep).
   – Send $(\mathsf{pot.deposit.end}, sid, dep)$ to $\mathcal{V}$.

5. On input $(\mathsf{pot.transfer.ini}, sid, ep, \sigma)$ from $\mathcal{B}$:
   – Abort if $(sid, ep', \langle m_n \rangle_{n=1}^{N}, 1)$ for $ep' = ep$ is not stored.
   – Abort if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_b)$ and $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}, ct_v)$ are not stored, or if $ct_b \neq ct_v$.
   – Abort if $\sigma \notin [1, N]$, or if $dep' < p_\sigma$, where dep' is stored in $(sid, dep', \mathsf{Tbl}_{st})$.
   – Create a fresh qid and store $(qid, ep, \sigma, m_\sigma)$.
   – Send $(\mathsf{pot.transfer.sim}, sid, qid, ep)$ to $\mathcal{S}$.

S. On input $(\mathsf{pot.transfer.rep}, sid, qid)$ from $\mathcal{S}$:
   – Abort if $(qid, ep, \sigma, m_\sigma)$ is not stored.
   – Set $dep' \leftarrow dep' - p_\sigma$, increment v_σ for the entry $[\sigma, v_\sigma]$ in Tbl_{st}, and update $(sid, dep', \mathsf{Tbl}_{st})$.
   – Delete the record $(qid, ep, \sigma, m_\sigma)$.
   – Send $(\mathsf{pot.transfer.end}, sid, m_\sigma)$ to $\mathcal{B}$.

6. On input $(\mathsf{pot.revealstatistic.ini}, sid, \mathsf{ST})$ from $\mathcal{B}$:
   – Abort if $(sid, dep', \mathsf{Tbl}_{st})$ is not stored.
   – Abort if $\mathsf{ST} \notin \Psi$.
   – Set $v \leftarrow \mathsf{ST}(\mathsf{Tbl}_{st})$.
   – Create a fresh qid and store (qid, v, ST).
   – Send $(\mathsf{pot.revealstatistic.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{pot.revealstatistic.rep}, sid, qid)$ from $\mathcal{S}$:
   – Abort if (qid, v, ST) is not stored.
   – Delete the record (qid, v, ST).
   – Send $(\mathsf{pot.revealstatistic.end}, sid, v, \mathsf{ST})$ to $\mathcal{V}$.

5 Building Blocks of Our Construction

Ideal Functionality $\mathcal{F}_{\mathrm{AUT}}$. Our protocol uses the functionality $\mathcal{F}_{\mathrm{AUT}}$ for an authenticated channel in [12]. $\mathcal{F}_{\mathrm{AUT}}$ interacts with a sender $\mathcal{T}$ and a receiver $\mathcal{R}$, and consists of one interface aut.send. $\mathcal{T}$ uses aut.send to send a message m to $\mathcal{F}_{\mathrm{AUT}}$. $\mathcal{F}_{\mathrm{AUT}}$ leaks m to the simulator $\mathcal{S}$ and, after receiving a response from $\mathcal{S}$, $\mathcal{F}_{\mathrm{AUT}}$ sends m to $\mathcal{R}$. $\mathcal{S}$ cannot modify m. The session identifier sid contains the identities of $\mathcal{T}$ and $\mathcal{R}$.

Ideal Functionality $\mathcal{F}_{\text{SMT}}$. Our protocol uses the functionality $\mathcal{F}_{\text{SMT}}$ for secure message transmission described in [12]. $\mathcal{F}_{\text{SMT}}$ interacts with a sender $\mathcal{T}$ and a receiver $\mathcal{R}$, and consists of one interface $\mathsf{smt.send}$. $\mathcal{T}$ uses the $\mathsf{smt.send}$ interface to send a message m to $\mathcal{F}_{\text{SMT}}$. $\mathcal{F}_{\text{SMT}}$ leaks $l(m)$, where $l : \mathcal{M} \to \mathbb{N}$ is a function that leaks the message length, to the simulator $\mathcal{S}$. After receiving a response from $\mathcal{S}$, $\mathcal{F}_{\text{SMT}}$ sends m to $\mathcal{R}$. $\mathcal{S}$ cannot modify m. The session identifier sid contains the identities of $\mathcal{T}$ and $\mathcal{R}$.

Ideal Functionality $\mathcal{F}_{\text{NIC}}$. Our protocol uses the functionality $\mathcal{F}_{\text{NIC}}$ for non-interactive commitments in [9]. $\mathcal{F}_{\text{NIC}}$ interacts with parties $\mathcal{P}_i$ and consists of the following interfaces:

1. Any party $\mathcal{P}_i$ uses the $\mathsf{com.setup}$ interface to set up the functionality.
2. Any party $\mathcal{P}_i$ uses the $\mathsf{com.commit}$ interface to send a message cm and obtain a commitment $ccom$ and an opening $copen$. A commitment $ccom$ is a tuple $(ccom'$, $cparcom, \mathsf{COM.Verify})$, where $ccom'$ is the commitment, $cparcom$ are the public parameters, and $\mathsf{COM.Verify}$ is the verification algorithm.
3. Any party $\mathcal{P}_i$ uses the $\mathsf{com.validate}$ interface to send a commitment $ccom$ to check that $ccom$ contains the correct public parameters and verification algorithm.
4. Any party $\mathcal{P}_i$ uses the $\mathsf{com.verify}$ interface to send $(ccom, cm, copen)$ in order to verify that $ccom$ is a commitment to the message cm with the opening $copen$.

$\mathcal{F}_{\text{NIC}}$ can be realized by a perfectly hiding commitment scheme, such as Pedersen commitments [9]. In [9], a method is described to use $\mathcal{F}_{\text{NIC}}$ in order to ensure that a party sends the same input cm to several ideal functionalities. For this purpose, the party first uses $\mathsf{com.commit}$ to get a commitment $ccom$ to cm with opening $copen$. Then the party sends $(ccom, cm, copen)$ as input to each of the functionalities, and each functionality runs $\mathsf{COM.Verify}$ to verify the commitment. Finally, other parties in the protocol receive the commitment $ccom$ from each of the functionalities and use the $\mathsf{com.validate}$ interface to validate $ccom$. Then, if $ccom$ received from all the functionalities is the same, the binding property provided by $\mathcal{F}_{\text{NIC}}$ ensures that all the functionalities received the same input cm. When using $\mathcal{F}_{\text{NIC}}$, it is needed to work in the $\mathcal{F}_{\text{NIC}}||\mathcal{S}_{\text{NIC}}$-hybrid model, where $\mathcal{S}_{\text{NIC}}$ is any simulator for a construction that realizes $\mathcal{F}_{\text{NIC}}$.

Ideal Functionality $\mathcal{F}_{\text{ZK}}^{R}$. Let R be a polynomial time computable binary relation. For tuples $(wit, ins) \in R$ we call wit the witness and ins the instance. Our protocol uses the ideal functionality $\mathcal{F}_{\text{ZK}}^{R}$ for zero-knowledge in [12]. $\mathcal{F}_{\text{ZK}}^{R}$ is parameterized by a description of a relation R, runs with a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, and consists of one interface $\mathsf{zk.prove}$. $\mathcal{P}$ uses $\mathsf{zk.prove}$ to send a witness wit and an instance ins to $\mathcal{F}_{\text{ZK}}^{R}$. $\mathcal{F}_{\text{ZK}}^{R}$ checks whether $(wit, ins) \in R$, and, in that case, sends the instance ins to $\mathcal{V}$. The simulator $\mathcal{S}$ learns ins but not wit. In our POT protocol, we use relations that include commitments as part of the instance, while the committed value and the opening are part of the witness. The relation uses the verification algorithm of the commitment scheme to check correctness of the commitment. This allows us to use the method described in [9] to ensure that an input to $\mathcal{F}_{\text{ZK}}^{R}$ is equal to the input of other functionalities.

Ideal Functionality $\mathcal{F}_{\text{OT}}$. Our protocol uses the ideal functionality $\mathcal{F}_{\text{OT}}$ for oblivious transfer. $\mathcal{F}_{\text{OT}}$ interacts with a sender $\mathcal{E}$ and a receiver $\mathcal{R}$, and consists of three interfaces $\mathsf{ot.init}$, $\mathsf{ot.request}$ and $\mathsf{ot.transfer}$.

1. $\mathcal{E}$ uses the ot.init interface to send the messages $\langle m_n \rangle_{n=1}^N$ to $\mathcal{F}_{\mathrm{OT}}$. $\mathcal{F}_{\mathrm{OT}}$ stores $\langle m_n \rangle_{n=1}^N$ and sends N to $\mathcal{R}$. The simulator $\mathcal{S}$ also learns N.
2. $\mathcal{R}$ uses the ot.request interface to send an index $\sigma \in [1, N]$, a commitment $ccom_\sigma$ and an opening $copen_\sigma$ to $\mathcal{F}_{\mathrm{OT}}$. $\mathcal{F}_{\mathrm{OT}}$ parses the commitment $ccom_\sigma$ as $(cparcom, com_\sigma, \mathsf{COM.Verify})$ and verifies the commitment by running $\mathsf{COM.Verify}$. $\mathcal{F}_{\mathrm{OT}}$ stores $[\sigma, ccom_\sigma]$ and sends $ccom_\sigma$ to $\mathcal{E}$.
3. $\mathcal{E}$ uses the ot.transfer interface to send a commitment $ccom_\sigma$ to $\mathcal{F}_{\mathrm{OT}}$. If a tuple $[\sigma, ccom_\sigma]$ is stored, $\mathcal{F}_{\mathrm{OT}}$ sends the message m_σ to $\mathcal{R}$.

$\mathcal{F}_{\mathrm{OT}}$ is similar to existing functionalities for OT [11], except that it receives a commitment $ccom_\sigma$ to the index σ and an opening $copen_\sigma$ for that commitment. In addition, the transfer phase is split up into two interfaces ot.request and ot.transfer, so that $\mathcal{E}$ receives $ccom_\sigma$ in the request phase. These changes are needed to use in our POT protocol the method in [9] to ensure that, when purchasing an item, the buyer sends the same index σ to $\mathcal{F}_{\mathrm{OT}}$ and to other functionalities. It is generally easy to modify existing UC OT protocols so that they realize our functionality $\mathcal{F}_{\mathrm{OT}}$.

Ideal Functionality $\mathcal{F}_{\mathrm{CD}}$. Our protocol uses the ideal functionality $\mathcal{F}_{\mathrm{CD}}$ for a committed database in [8]. $\mathcal{F}_{\mathrm{CD}}$ interacts with a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, and consists of three interfaces cd.setup, cd.read and cd.write.

1. $\mathcal{V}$ uses the cd.setup interface to initialize Tbl_{cd}. $\mathcal{F}_{\mathrm{CD}}$ stores Tbl_{cd} and sends Tbl_{cd} to $\mathcal{P}$ and to the simulator $\mathcal{S}$.
2. $\mathcal{P}$ uses cd.read to send a position i and a value v_r to $\mathcal{F}_{\mathrm{CD}}$, along with commitments and openings $(ccom_i, copen_i)$ and $(ccom_r, copen_r)$ to the position and value respectively. $\mathcal{F}_{\mathrm{CD}}$ verifies the commitments and checks that there is an entry $[i, v_r]$ in the table Tbl_{cd}. In that case, $\mathcal{F}_{\mathrm{CD}}$ sends $ccom_i$ and $ccom_r$ to $\mathcal{V}$. $\mathcal{S}$ also learns $ccom_i$ and $ccom_r$.
3. $\mathcal{P}$ uses cd.write to send a position i and a value v_w to $\mathcal{F}_{\mathrm{CD}}$, along with commitments and openings $(ccom_i, copen_i)$ and $(ccom_w, copen_w)$ to the position and value respectively. $\mathcal{F}_{\mathrm{CD}}$ verifies the commitments and updates Tbl_{cd} to store v_w at position i. $\mathcal{F}_{\mathrm{CD}}$ sends $ccom_i$ and $ccom_w$ to $\mathcal{V}$. $\mathcal{S}$ also learns $ccom_i$ and $ccom_w$.

Basically, $\mathcal{F}_{\mathrm{CD}}$ allows $\mathcal{P}$ to prove to $\mathcal{V}$ that two commitments $ccom_i$ and $ccom_r$ commit to a position and value that are read from a table, and that two commitments $ccom_i$ and $ccom_w$ commit to a position and value that are written into the table. In [8], an efficient construction for $\mathcal{F}_{\mathrm{CD}}$ based on hiding vector commitments [20,15] is proposed. In our POT protocol, $\mathcal{F}_{\mathrm{CD}}$ is used to store and update the deposit of the buyer and the counters of the number of purchases for each of the item categories.

Ideal Functionality $\mathcal{F}_{\mathrm{NHCD}}$. Our protocol uses the ideal functionality $\mathcal{F}_{\mathrm{NHCD}}$ for a non-hiding committed database in [21]. $\mathcal{F}_{\mathrm{NHCD}}$ interacts with a party $\mathcal{P}_0$ and a party $\mathcal{P}_1$, and consists of three interfaces nhcd.setup, nhcd.prove and nhcd.write.

1. $\mathcal{P}_1$ uses nhcd.setup to send a table Tbl_{nhcd} with N entries of the form $[i, v]$ (for $i = 0$ to N) to $\mathcal{F}_{\mathrm{NHCD}}$. $\mathcal{F}_{\mathrm{NHCD}}$ stores Tbl_{nhcd} and sends Tbl_{nhcd} to $\mathcal{P}_0$. The simulator $\mathcal{S}$ also learns Tbl_{nhcd}.

2. $\mathcal{P}_b$ ($b \in [0,1]$) uses nhcd.prove to send a position i and a value v_r to $\mathcal{F}_{\mathrm{NHCD}}$, along with commitments and openings $(ccom_i, copen_i)$ and $(ccom_r, copen_r)$ to the position and value respectively. $\mathcal{F}_{\mathrm{NHCD}}$ verifies the commitments and checks that there is an entry $[i, v_r]$ in the table Tbl_{nhcd}. In that case, $\mathcal{F}_{\mathrm{NHCD}}$ sends $ccom_i$ and $ccom_r$ to $\mathcal{P}_{1-b}$. The simulator $\mathcal{S}$ also learns $ccom_i$ and $ccom_r$.
3. $\mathcal{P}_1$ uses nhcd.write to send a position i and a value v_w to $\mathcal{F}_{\mathrm{NHCD}}$. $\mathcal{F}_{\mathrm{NHCD}}$ updates Tbl_{nhcd} to contain value v_w at position i and sends i and v_w to $\mathcal{P}_0$. The simulator $\mathcal{S}$ also learns i and v_w.

$\mathcal{F}_{\mathrm{NHCD}}$ is similar to the functionality $\mathcal{F}_{\mathrm{CD}}$ described above. The main difference is that the contents of the table Tbl_{nhcd} are known by both parties. For this reason, both parties can invoke the nhcd.prove interface to prove that two commitments $ccom_i$ and $ccom_r$ commit to a position and value stored in Tbl_{nhcd}. In addition, the interface nhcd.write reveals the updates to Tbl_{nhcd} made by $\mathcal{P}_1$ to $\mathcal{P}_0$. In [21], an efficient construction for $\mathcal{F}_{\mathrm{NHCD}}$ based on non-hiding vector commitments is proposed. In our POT protocol, $\mathcal{F}_{\mathrm{NHCD}}$ will be used by the seller, acting as $\mathcal{P}_1$, to store and update the prices of items. The buyer, acting as $\mathcal{P}_0$, uses the nhcd.prove interface to prove to the seller that the correct price for the item purchased is used.

The full description of the ideal functionalities is given in Section A of the supplementary material.

6 Construction Π_{POTS} for $\mathcal{F}_{\mathrm{POTS}}$

Intuition. For each epoch ep, $\mathcal{V}$ and $\mathcal{B}$ use a new instance of $\mathcal{F}_{\mathrm{OT}}$. In the pot.init interface, $\mathcal{V}$ uses ot.init to create a new instance of $\mathcal{F}_{\mathrm{OT}}$ on input the messages $\langle m_n \rangle_{n=1}^{N}$. In the pot.transfer interface, $\mathcal{B}$ uses ot.request on input an index σ and receives the message m_σ through the ot.transfer interface.

To construct a POT protocol based on $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{V}$ must set the prices, and $\mathcal{B}$ must make deposits and pay for the messages obtained. Additionally, our POT protocol allows $\mathcal{V}$ to receive aggregate statistics about the purchases.

Prices. To set prices, $\mathcal{V}$ uses $\mathcal{F}_{\mathrm{NHCD}}$. In the pot.setupprices interface, the seller $\mathcal{V}$ uses nhcd.setup to create an instance of $\mathcal{F}_{\mathrm{NHCD}}$ on input a list of prices $\langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}$, which are stored in the table Tbl_{nhcd} in $\mathcal{F}_{\mathrm{NHCD}}$. In the pot.updateprice interface, $\mathcal{V}$ uses nhcd.write to update a price in the table Tbl_{nhcd}.

Deposits. $\mathcal{F}_{\mathrm{CD}}$ is used to store the current funds dep' of $\mathcal{B}$. In the pot.init interface, $\mathcal{V}$ uses cd.setup to create an instance of $\mathcal{F}_{\mathrm{CD}}$ on input a table Tbl_{cd} that contains a 0 at every position. The position 0 of Tbl_{cd} is used to store dep'. In the pot.deposit interface, $\mathcal{B}$ makes deposits dep to $\mathcal{V}$, which are added to the existing funds dep'. (To carry out the payment of dep, $\mathcal{V}$ and $\mathcal{B}$ use a payment mechanism outside the POT protocol.) $\mathcal{B}$ uses $\mathcal{F}_{\mathrm{ZK}}^{R_{dep}}$ to prove in zero-knowledge to $\mathcal{V}$ that the deposit is updated correctly as $dep' \leftarrow dep' + dep$. The interfaces cd.read and cd.write are used to read dep' and to write the updated value of dep' into Tbl_{cd}.

Payments. In the pot.transfer interface, $\mathcal{B}$ must subtract the price p_σ for the purchased message m_σ from the current funds dep'. $\mathcal{B}$ uses nhcd.prove to read the correct price p_σ from Tbl_{nhcd}. Then $\mathcal{B}$ uses $\mathcal{F}_{\mathrm{ZK}}^{R_{trans}}$ to prove in zero-knowledge that $dep' \leftarrow$

$dep' - p_\sigma$. The interfaces cd.read and cd.write of $\mathcal{F}_{\mathrm{CD}}$ are used to read dep' and to write the updated value of dep' into Tbl_{cd}.

Statistics. $\mathcal{F}_{\mathrm{CD}}$ is used to store counters on the number of purchases of each item category in the positions $[1, \mathcal{N}_{max}]$ of table Tbl_{cd}. In the pot.transfer interface, $\mathcal{B}$ uses cd.read to read the table entry $[\sigma, count_1]$ in Tbl_{cd}, where σ is the index of the message purchased m_σ and $count_1$ is the counter for that category. $\mathcal{B}$ computes $count_2 \leftarrow count_1 + 1$ and uses $\mathcal{F}_{\mathrm{ZK}}^{R_{count}}$ to prove in zero-knowledge that the counter is correctly incremented. Then $\mathcal{B}$ uses cd.write to write the entry $[\sigma, count_2]$ in Tbl_{cd}. In the pot.revealstatistic interface, $\mathcal{B}$ uses $\mathcal{F}_{\mathrm{ZK}}^{R_{\mathsf{ST}}}$ to prove in zero-knowledge to $\mathcal{V}$ that a statistic v is the result of evaluating a function ST on input Tbl_{cd}. For this purpose, $\mathcal{B}$ uses cd.read to read the required table entries in Tbl_{cd}.

Construction Π_{POTS}. Π_{POTS} is parameterised with a maximum number of messages $\mathcal{N}_{max}$, a message space $\mathcal{M}$, a maximum deposit value dep_{max}, a maximum price $\mathcal{P}_{max}$, and a universe of statistics Ψ that consists of ppt algorithms. Π_{POTS} uses the ideal functionalities $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$ and $\mathcal{F}_{\mathrm{NHCD}}$. We omit some abortion conditions or some of the messages used to invoke functionalities, which are depicted in Section B of the supplementary material.

1. On input (pot.init.ini, sid, ep, $\langle m_n \rangle_{n=1}^{N}$), $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - If this is the first execution of this interface, $\mathcal{V}$ and $\mathcal{B}$ set up $\mathcal{F}_{\mathrm{CD}}$ as follows:
     - $\mathcal{V}$ sets a table Tbl_{cd} of $\mathcal{N}_{max}$ entries of the form $[i, 0]$ for $i = 0$ to $\mathcal{N}_{max}$. $\mathcal{V}$ uses cd.setup to send Tbl_{cd} to a new instance of $\mathcal{F}_{\mathrm{CD}}$.
     - $\mathcal{B}$ receives Tbl_{cd} from $\mathcal{F}_{\mathrm{CD}}$ and stores (sid, Tbl_{cd}). Then $\mathcal{B}$ sends the message *setup* to $\mathcal{V}$ via $\mathcal{F}_{\mathrm{AUT}}$, so that $\mathcal{V}$ continues the protocol execution.
   - $\mathcal{V}$ uses ot.init to send the messages $\langle m_n \rangle_{n=1}^{N}$ to a new instance of $\mathcal{F}_{\mathrm{OT}}$ with session identifier $sid_{\mathrm{OT}} \leftarrow (sid, ep)$.
   - $\mathcal{B}$ receives the messages from $\mathcal{F}_{\mathrm{OT}}$, stores (sid, ep, N), and outputs the message (pot.init.end, sid, ep, N).
2. On input (pot.setupprices.ini, sid, $\langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}$), $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - For $n = 1$ to $\mathcal{N}_{max}$, $\mathcal{V}$ sets a table Tbl_{nhcd} with entries $[n, p_n]$ and uses nhcd.setup to send Tbl_{nhcd} to a new instance of $\mathcal{F}_{\mathrm{NHCD}}$.
   - $\mathcal{B}$ receives Tbl_{nhcd} from $\mathcal{F}_{\mathrm{NHCD}}$, parses Tbl_{nhcd} as $[n, p_n]$, for $n = 1$ to $\mathcal{N}_{max}$, stores $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$ and outputs (pot.setupprices.end, sid, $\langle p_n \rangle_{n=1}^{\mathcal{N}_{max}}$).
3. On input (pot.updateprice.ini, sid, n, p), $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - $\mathcal{V}$ uses nhcd.write to send the index n and the price p to $\mathcal{F}_{\mathrm{NHCD}}$.
   - $\mathcal{B}$ receives n and p from $\mathcal{F}_{\mathrm{NHCD}}$, updates the stored tuple $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$ and outputs (pot.updateprice.end, sid, n, p).
4. On input (pot.deposit.ini, sid, dep), $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - If this is the first execution of the deposit interface, $\mathcal{B}$ uses com.setup to create a new instance of $\mathcal{F}_{\mathrm{NIC}}$, sets $dep_1 \leftarrow 0$ and uses com.commit to get a commitment $ccom_{dep_1}$ to dep_1 with opening $copen_{dep_1}$ from $\mathcal{F}_{\mathrm{NIC}}$. Otherwise, $\mathcal{B}$ takes the stored commitment to the deposit $(sid, ccom_{dep_2}, copen_{dep_2})$, sets $ccom_{dep_1} \leftarrow ccom_{dep_2}$ and $copen_{dep_1} \leftarrow copen_{dep_2}$, and sets $dep_1 \leftarrow v$ where $[0, v]$ is the deposit stored in the table (sid, Tbl_{cd}).

- $\mathcal{B}$ sets $dep_2 \leftarrow dep_1 + dep$ and uses com.commit to get commitments and openings $(ccom_{dep}, copen_{dep})$ and $(ccom_{dep_2}, copen_{dep_2})$ to dep and dep_2.
- $\mathcal{B}$ sets a witness wit_{dep} as $(dep, copen_{dep}, dep_1, copen_{dep_1}, dep_2, copen_{dep_2})$ and an instance ins_{dep} as $(cparcom, ccom_{dep}, ccom_{dep_1}, ccom_{dep_2})$, stores $(sid, wit_{dep}, ins_{dep})$, and uses zk.prove to send wit_{dep} and ins_{dep} to $\mathcal{F}_{\mathrm{ZK}}^{R_{dep}}$, where relation R_{dep} is

$$\begin{aligned} R_{dep} = \{ & (wit_{dep}, ins_{dep}) : \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{dep}, dep, copen_{dep}) \wedge \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{dep_1}, dep_1, copen_{dep_1}) \wedge \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{dep_2}, dep_2, copen_{dep_2}) \wedge \\ & dep_2 = dep + dep_1 \ \wedge dep_2 \in [0, dep_{max}] \} \end{aligned}$$

- $\mathcal{V}$ receives $ins_{dep} = (cparcom, ccom_{dep}, ccom_{dep_1}, ccom_{dep_2})$ from $\mathcal{F}_{\mathrm{ZK}}^{R_{dep}}$. If this is the first execution of the deposit interface, $\mathcal{V}$ invokes com.setup of $\mathcal{F}_{\mathrm{NIC}}$ and then uses com.validate to validate $ccom_{dep_1}$. Otherwise, $\mathcal{V}$ takes the stored commitment $(sid, ccom'_{dep_2})$ (which commits to the old value of the deposit) and aborts if $ccom'_{dep_2} \neq ccom_{dep_1}$, because this means that the buyer used an incorrect commitment $ccom_{dep_1}$ to the old value of the deposit.
- $\mathcal{V}$ uses com.validate to validate $ccom_{dep_2}$ and $ccom_{dep}$ and stores the tuple (sid, ins_{dep}).
- $\mathcal{V}$ uses aut.send to send *writedeposit* to $\mathcal{B}$, so that $\mathcal{B}$ continues the protocol.
- If this is the first execution of the deposit interface, $\mathcal{B}$ uses com.commit to get a commitment and opening $(ccom_0, copen_0)$ to 0, which is the position where the deposit is stored in the table of $\mathcal{F}_{\mathrm{CD}}$.
- $\mathcal{B}$ stores $(sid, ccom_{dep_2}, copen_{dep_2})$ and uses the cd.write interface to send $(ccom_0, 0, copen_0)$ and $(ccom_{dep_2}, dep_2, copen_{dep_2})$ in order to write an entry $[0, dep_2]$ into the table of $\mathcal{F}_{\mathrm{CD}}$.
- $\mathcal{V}$ receives $(\mathsf{cd.write.end}, sid, ccom_0, ccom_{dep_2})$ from $\mathcal{F}_{\mathrm{CD}}$.
- $\mathcal{V}$ aborts if the commitment $ccom_{dep_2}$ stored in (sid, ins_{dep}) is not the same as that received from $\mathcal{F}_{\mathrm{CD}}$. If this is not the first execution of the deposit interface, $\mathcal{V}$ aborts if $(sid, ccom_0)$ is not the same as the commitment received from $\mathcal{F}_{\mathrm{CD}}$.
- $\mathcal{V}$ uses aut.send to send a message *revealdeposit* to $\mathcal{B}$.
- $\mathcal{B}$ updates (sid, Tbl_{cd}) with $[0, dep_2]$.
- If this is the first execution of the deposit interface, $\mathcal{B}$ uses smt.send to send $\langle dep, copen_{dep}, 0, copen_0, dep_1, copen_{dep_1} \rangle$ to $\mathcal{V}$, and $\mathcal{V}$ uses com.verify to verify $(ccom_0, 0, copen_0)$ and $(ccom_{dep_1}, dep_1, copen_{dep_1})$. Else the buyer $\mathcal{B}$ uses the smt.send interface to send $\langle dep, copen_{dep} \rangle$ to $\mathcal{V}$.
- $\mathcal{V}$ uses com.verify to verify $(ccom_{dep}, dep, copen_{dep})$.
- $\mathcal{V}$ outputs $(\mathsf{pot.deposit.end}, sid, dep)$.

5. On input $(\mathsf{pot.transfer.ini}, sid, ep, \sigma)$, $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - $\mathcal{B}$ retrieves the entry $[0, dep_1]$ from the table (sid, Tbl_{cd}) and the price p_σ from $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$, and sets $dep_2 \leftarrow dep_1 - p_\sigma$.
   - $\mathcal{B}$ retrieves the stored tuple $(sid, ccom_{dep_2}, copen_{dep_2})$ and sets $ccom_{dep_1} \leftarrow ccom_{dep_2}$ and $copen_{dep_1} \leftarrow copen_{dep_2}$.

- $\mathcal{B}$ uses com.commit to obtain from functionality $\mathcal{F}_{\mathrm{NIC}}$ the commitments and openings $(ccom_{dep_2}, copen_{dep_2})$ to dep_2, $(ccom_\sigma, copen_\sigma)$ to σ and $(ccom_{p_\sigma}, copen_{p_\sigma})$ to p_σ.
- $\mathcal{B}$ sets wit_{trans} as $(p_\sigma, copen_{p_\sigma}, dep_1, copen_{dep_1}, dep_2, copen_{dep_2})$ and the instance ins_{trans} as $(cparcom, ccom_{p_\sigma}, ccom_{dep_1}, ccom_{dep_2})$, stores the tuple $(sid, wit_{trans}, ins_{trans})$, and uses zk.prove to send wit_{trans} and ins_{trans} to $\mathcal{F}_{\mathrm{ZK}}^{R_{trans}}$, where R_{trans} is defined as follows

$$\begin{aligned} R_{trans} = \{ & (wit_{trans}, ins_{trans}) : \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{p_\sigma}, p_\sigma, copen_{p_\sigma}) \wedge \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{dep_1}, dep_1, copen_{dep_1}) \wedge \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{dep_2}, dep_2, copen_{dep_2}) \wedge \\ & dep_2 = dep_1 - p_\sigma \ \wedge \ dep_2 \in [0, dep_{max}] \} \end{aligned}$$

- $\mathcal{V}$ receives $ins_{trans} = (cparcom, ccom_{p_\sigma}, ccom_{dep_1}, ccom_{dep_2})$ from $\mathcal{F}_{\mathrm{ZK}}^{R_{trans}}$, uses com.validate to validate the commitments $ccom_{p_\sigma}$ and $ccom_{dep_2}$, and aborts if $ccom_{dep_1}$ is not equal to the stored commitment $(sid, ccom_{dep_2})$.
- $\mathcal{V}$ stores (sid, ins_{trans}) and uses aut.send to send a message *readprice* to $\mathcal{B}$, so that $\mathcal{B}$ continues the protocol execution.
- $\mathcal{B}$ uses nhcd.prove to send the position σ and the price p_σ to $\mathcal{F}_{\mathrm{NHCD}}$, along with their respective commitments and openings $(ccom_\sigma, copen_\sigma)$ and $(ccom_{p_\sigma}, copen_{p_\sigma})$.
- $\mathcal{V}$ receives $ccom_\sigma$ and $ccom_{p_\sigma}$ from $\mathcal{F}_{\mathrm{NHCD}}$, aborts if $ccom_{p_\sigma}$ is not equal to the commitment stored in (sid, ins_{trans}), uses com.validate to validate $ccom_\sigma$ and adds $ccom_\sigma$ to (sid, ins_{trans}).
- $\mathcal{V}$ uses aut.send to send the message *commitdeposit* to $\mathcal{B}$.
- $\mathcal{B}$ uses cd.write to send the position 0 and the deposit dep_2 into $\mathcal{F}_{\mathrm{CD}}$, along with $(ccom_0, copen_0)$ and $(ccom_{dep_2}, copen_{dep_2})$.
- $\mathcal{V}$ receives $ccom_0$ and $ccom_{dep_2}$ from $\mathcal{F}_{\mathrm{CD}}$, and aborts if the $ccom_{dep_2}$ in (sid, ins_{trans}) is not the same as that received from $\mathcal{F}_{\mathrm{CD}}$, or if $ccom_0$ received from $\mathcal{F}_{\mathrm{CD}}$ is not the same as $(sid, ccom_0)$ stored during the first execution of the deposit interface.
- $\mathcal{V}$ uses aut.send to send the message *commitcounter* to $\mathcal{B}$.
- $\mathcal{B}$ stores $(sid, ccom_{dep_2}, copen_{dep_2})$, updates (sid, Tbl_{cd}) with $[0, dep_2]$, then deletes $(sid, wit_{trans}, ins_{trans})$ and stores $(sid, wit_{count}, ins_{count})$.
- $\mathcal{B}$ retrieves $[\sigma, v]$ from Tbl_{cd}, sets $count_1 \leftarrow v$ and $count_2 \leftarrow count_1 + 1$, and uses com.commit to get from $\mathcal{F}_{\mathrm{NIC}}$ commitments and openings $(ccom_{count_1}, copen_{count_1})$ to $count_1$ and $(ccom_{count_2}, copen_{count_2})$ to $count_2$.
- $\mathcal{B}$ sets wit_{count} as $(\sigma, copen_\sigma, count_1, copen_{count_1}, count_2, copen_{count_2})$ and ins_{count} as $(cparcom, ccom_\sigma, ccom_{count_1}, ccom_{count_2})$, and uses zk.prove to

send wit_{count} and ins_{count} to $\mathcal{F}_{\mathrm{ZK}}^{R_{count}}$, where R_{count} is

$$\begin{aligned}R_{count} =\{(&wit_{count}, ins_{count}) :\\ &1 = \mathsf{COM.Verify}(cparcom, ccom_{\sigma}, \sigma, copen_{\sigma}) \wedge\\ &1 = \mathsf{COM.Verify}(cparcom, ccom_{count_1}, count_1, copen_{count_1}) \wedge\\ &1 = \mathsf{COM.Verify}(cparcom, ccom_{count_2}, count_2, copen_{count_2}) \wedge\\ &count_2 = count_1 + 1\}\end{aligned}$$

- $\mathcal{V}$ receives ins_{count} from $\mathcal{F}_{\mathrm{ZK}}^{R_{count}}$, aborts if the commitment $ccom_{\sigma}$ in (sid, ins_{trans}) is not equal to the one in ins_{count}, uses com.validate to validate the commitments $ccom_{count_1}$ and $ccom_{count_2}$, and stores (sid, ins_{count}).
- $\mathcal{V}$ uses aut.send to send the message *readcounter* to $\mathcal{B}$.
- $\mathcal{B}$ uses cd.read to send the position σ and the counter $count_1$ to $\mathcal{F}_{\mathrm{CD}}$, along with $(ccom_{\sigma}, copen_{\sigma})$ and $(ccom_{count_1}, copen_{count_1})$.
- $\mathcal{V}$ receives $ccom_{\sigma}$ and $ccom_{count_1}$ from $\mathcal{F}_{\mathrm{CD}}$ and aborts if those commitments are not equal to the ones stored in (sid, ins_{count}).
- $\mathcal{V}$ uses aut.send to send the message *writecounter* to $\mathcal{B}$.
- $\mathcal{B}$ uses cd.write to send the position σ and the value $count_2$ to $\mathcal{F}_{\mathrm{CD}}$, along with $(ccom_{\sigma}, copen_{\sigma})$ and $(ccom_{count_2}, copen_{count_2})$.
- $\mathcal{V}$ receives $ccom_{\sigma}$ and $ccom_{count_2}$ from $\mathcal{F}_{\mathrm{CD}}$ and aborts if those commitments are not equal to the ones stored in (sid, ins_{count}).
- $\mathcal{V}$ uses aut.send to send the message *transfer* to $\mathcal{B}$.
- $\mathcal{B}$ updates (sid, Tbl_{cd}) with $[\sigma, count_2]$ and uses ot.request to send the index σ along with $(ccom_{\sigma}, copen_{\sigma})$ to the instance of $\mathcal{F}_{\mathrm{OT}}$ with session identifier $sid_{\mathrm{OT}} = (sid, ep)$.
- $\mathcal{V}$ receives $ccom_{\sigma}$ from $\mathcal{F}_{\mathrm{OT}}$ and aborts if it is not equal to the one contained in ins_{count}.
- $\mathcal{V}$ uses ot.transfer to send $ccom_{\sigma}$ to the instance of $\mathcal{F}_{\mathrm{OT}}$ with session identifier $sid_{\mathrm{OT}} = (sid, ep)$.
- $\mathcal{B}$ receives m_{σ} from $\mathcal{F}_{\mathrm{OT}}$ and outputs $(\mathsf{pot.transfer.end}, sid, m_{\sigma})$.

6. On input $(\mathsf{pot.revealstatistic.ini}, sid, \mathsf{ST})$, $\mathcal{B}$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}$ takes the stored (sid, Tbl_{cd}) and computes $result \leftarrow \mathsf{ST}(\mathsf{Tbl}_{cd})$.
   - For all $i \in \mathbb{P}$, where $\mathbb{P}$ is the subset of positions i such that the entry $[i, v_i]$ in Tbl_{cd} was used by $\mathcal{B}$ to compute $result$, $\mathcal{B}$ and $\mathcal{V}$ do the following:
     - $\mathcal{B}$ uses com.commit to get the commitments and openings $(ccom_i, copen_i)$ to i and $(ccom_{v_i}, copen_{v_i})$ to v_i and stores $(sid, ccom_i, ccom_{v_i})$. Then $\mathcal{B}$ uses cd.read to read $(ccom_i, i, copen_i)$ and $(ccom_{v_i}, v_i, copen_{v_i})$ from $\mathcal{F}_{\mathrm{CD}}$.
     - $\mathcal{V}$ receives $(ccom_i, ccom_{v_i})$ from $\mathcal{F}_{\mathrm{CD}}$, uses com.validate to validate the commitments $ccom_i$ and $ccom_{v_i}$, and uses aut.send to send the following message $\langle OK, ccom_i, ccom_{v_i}\rangle$ to $\mathcal{B}$.
   - $\mathcal{B}$ sets $wit_{\mathsf{ST}} \leftarrow (\langle i, copen_i, v_i, copen_{v_i}\rangle_{\forall i})$ and $ins_{\mathsf{ST}} \leftarrow (result, cparcom, \langle ccom_i, ccom_{v_i}\rangle_{\forall i})$, and uses zk.prove to send wit_{ST} and ins_{ST} to $\mathcal{F}_{\mathrm{ZK}}^{R_{\mathsf{ST}}}$, where

the relation R_{ST} is

$$\begin{aligned} R_{\mathsf{ST}} =\{ & (wit_{\mathsf{ST}}, ins_{\mathsf{ST}}) : \\ & [1 = \mathsf{COM.Verify}(cparcom, ccom_i, i, copen_i) \wedge \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{v_i}, v_i, copen_{v_i}) \,]_{\forall i \in \mathbb{P}} \wedge \\ & result = \mathsf{ST}(\langle i, v_i \rangle_{\forall i \in \mathbb{P}}) \, \} \end{aligned}$$

– $\mathcal{V}$ receives ins_{ST} from $\mathcal{F}_{\mathrm{ZK}}^{R_{\mathsf{ST}}}$ and aborts if the commitments received from $\mathcal{F}_{\mathrm{CD}}$ are not the same as those in ins_{ST}.
– $\mathcal{V}$ outputs $(\mathsf{pot.revealstatistic.end}, sid, result, \mathsf{ST})$.

Theorem 1. *Construction Π_{POTS} realizes functionality $\mathcal{F}_{\mathrm{POTS}}$ in the ($\mathcal{F}_{\mathrm{AUT}}, \mathcal{F}_{\mathrm{SMT}}, \mathcal{F}_{\mathrm{NIC}} || \mathcal{S}_{\mathrm{NIC}}, \mathcal{F}_{\mathrm{ZK}}^{R}, \mathcal{F}_{\mathrm{OT}}, \mathcal{F}_{\mathrm{CD}}, \mathcal{F}_{\mathrm{NHCD}}$)-hybrid model.*

We prove this theorem in Section C of the supplementary material.

Instantiation and efficiency analysis. In previous work [1,6,23,24,22,2], the computation and communication cost of POT protocols is dominated by the cost of the underlying OT scheme. This is also the case for Π_{POTS}. However, Π_{POTS} has the advantage that it can be instantiated with any OT protocol that realizes $\mathcal{F}_{\mathrm{OT}}$. The OT schemes used to construct UC-secure POT schemes [23], and other UC-secure OT schemes are suitable. Moreover, when new more efficient OT schemes are available, they can also be used to instantiate Π_{POTS}.

We also note that the overhead introduced by $\mathcal{F}_{\mathrm{NIC}}$ to allow the modular design of Π_{POTS} is small. $\mathcal{F}_{\mathrm{NIC}}$ can be instantiated with a perfectly hiding commitment scheme, such as Pedersen commitments [9]. Therefore, the overhead consists in computing a commitment to each of the values that need to be sent to more than one functionality, and ZK proofs of the opening of those commitments.

As discussed above, to construct POT from an OT scheme, $\mathcal{V}$ must set the prices, and $\mathcal{B}$ must make deposits and pay for the messages obtained. Additionally, our POT protocol allows $\mathcal{V}$ to receive aggregate statistics about the purchases. For these tasks, Π_{POTS} uses $\mathcal{F}_{\mathrm{NHCD}}$ and $\mathcal{F}_{\mathrm{CD}}$. These functionalities can be instantiated with a non-hiding and hiding VC scheme respectively [8,21], equipped with ZK proofs of an opening for a position of the vector. In [8,21], concrete instantiations based on the Diffie-Hellman Exponent (DHE) assumption are provided. These instantiations involve a common reference string that grows linearly with the length of the committed vector, which in Π_{POTS} is the number N of messages. A non-hiding VC and a hiding VC commit to the tables Tbl_{nhcd} and Tbl_{cd} respectively. The vector commitments, as well as openings for each position of the vector, are of size independent of N. The computation of a commitment and of an opening grows linearly with N. However, when the committed vector changes, both the vector commitment and the openings can be updated with cost independent of N. Therefore, they can be updated and reused throughout the protocol, yielding amortized cost independent of N. The ZK proofs of VC openings offer computation and communication cost independent of N. Therefore, with this instantiation, Π_{POTS} remains efficient when the number N of messages is large.

We compare below Π_{POTS} to the UC-secure scheme in [23], but we note that this comparison would be similar for other full-simulation secure POT protocols. We can conclude that Π_{POTS} provides additional functionalities like dynamic pricing and aggregated statistics with cost similar to POT protocols that do not provide them.

Prices. In [23], in the initialization phase, $\mathcal{V}$ encrypts the messages and, for each message, $\mathcal{V}$ computes a signature that binds a ciphertext to the price of the encrypted message. This implies that one signature per message is sent from $\mathcal{V}$ to $\mathcal{B}$, and thus the cost grows linearly with N. In Π_{POTS}, $\mathcal{F}_{\mathrm{NHCD}}$ is used, which can be instantiated with a non-hiding VC scheme. In this instantiation, only one vector commitment, which commits to a vector that contains the list of prices, needs to be sent from $\mathcal{V}$ to $\mathcal{B}$. Nevertheless, adding the size of the common reference string, the cost also grows linearly with N.
However, non-hiding VC schemes provide dynamic pricing at no extra cost. The vector commitment can be updated with cost independent of N. With a signature scheme, $\mathcal{V}$ could also provide a new signature on the price with cost independent of N. However, $\mathcal{V}$ needs to revoke the signature on the old price. The need of a signature revocation mechanism makes dynamic pricing costly in this case.

Deposit. In [23], in the deposit phase, $\mathcal{B}$ sends a commitment to the new value of the deposit and a ZK proof that the deposit is updated. In Π_{POTS}, $\mathcal{F}_{\mathrm{CD}}$ is used, which can be instantiated with a hiding VC that stores the deposit at position 0. The size of commitments, as well as the cost of a ZK proof of deposit updated, does not depend on N in both cases. However, the common reference string (crs) of the VC scheme grows linearly with N. (We recall that $\mathcal{F}_{\mathrm{CD}}$ not only stores the deposit but also the N counters of purchases.) By applying the UC with joint state theorem [14], it could be possible to share crs for the DHE instantiations of the non-hiding and hiding VC schemes, but this affects the modularity of Π_{POTS}.

Payment. In [23], $\mathcal{B}$ proves in ZK that the price of the purchased message is subtracted from her current funds. This involves a ZK proof of signature possession, to prove that the correct price is used, and a ZK proof of commitment opening, to prove that the correct value of the deposit is used. The cost of these proofs is independent of N. In Π_{POTS}, when using non-hiding and hiding VC schemes to instantiate $\mathcal{F}_{\mathrm{NHCD}}$ and $\mathcal{F}_{\mathrm{CD}}$, we need two ZK proofs of a vector commitment opening, one for the non-hiding VC scheme (for the price) and one for the hiding VC scheme (for the deposit). The amortized cost of those ZK proofs is also independent of N. The cost of a ZK proof of commitment opening for the DHE instantiation is similar to the ZK proof of signature possession in [23].

Statistics. Unlike [23], Π_{POTS} allows $\mathcal{V}$ to get aggregate statistics about the purchases of $\mathcal{B}$. $\mathcal{F}_{\mathrm{CD}}$ stores the counters of the number of purchases for each category. With the instantiation based on a hiding VC scheme, updating the counters and reading them to compute a statistic involves again ZK proofs of the opening of positions of a VC, whose amortized computation and communication cost is independent of N.

Aggregate statistics about multiple buyers. Π_{POTS} allows $\mathcal{V}$ to gather statistics about the purchases of each buyer separately. Nonetheless, $\mathcal{V}$ is possibly more interested in gathering aggregate statistics about multiple buyers. This is also appealing to better protect buyer's privacy. Fortunately, Π_{POTS} enables this possibility. The functionalities $\mathcal{F}_{\mathrm{CD}}$ used in the execution of Π_{POTS} between $\mathcal{V}$ and each of the buyers can be used to run a secure multiparty computation (MPC) protocol for the required statistic. In this protocol, each buyer reads from $\mathcal{F}_{\mathrm{CD}}$ the counters needed for the statistic. We note that $\mathcal{F}_{\mathrm{CD}}$ provides commitments to the counters read. These commitments can easily be plugged into existing commit-and-prove MPC protocols [13] to run an MPC between the seller and the buyers. We note the previous POT protocols do not provide this possibility because there buyers do not have any means to prove what they purchased before. $\mathcal{F}_{\mathrm{CD}}$ acts as a ZK data structure that stores information about what buyers have proven in zero-knowledge, so that this information can be reused in subsequent ZK proofs.

Acknowledgements. This research is supported by the Luxembourg National Research Fund (FNR) CORE project "Stateful Zero-Knowledge" (Project code: C17/11650748).

References

1. Aiello, B., Ishai, Y., Reingold, O.: Priced oblivious transfer: How to sell digital goods. In: International Conference on the Theory and Applications of Cryptographic Techniques. pp. 119–135. Springer (2001)
2. Biesmans, W., Balasch, J., Rial, A., Preneel, B., Verbauwhede, I.: Private mobile pay-tv from priced oblivious transfer. IEEE Transactions on Information Forensics and Security **13**(2), 280–291 (2018)
3. Blazy, O., Chevalier, C., Germouty, P.: Adaptive oblivious transfer and generalization. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 217–247. Springer (2016)
4. Camenisch, J., Dubovitskaya, M., Enderlein, R.R., Neven, G.: Oblivious transfer with hidden access control from attribute-based encryption. In: International Conference on Security and Cryptography for Networks. pp. 559–579. Springer (2012)
5. Camenisch, J., Dubovitskaya, M., Neven, G.: Oblivious transfer with access control. In: Proceedings of the 16th ACM conference on Computer and communications security. pp. 131–140. ACM (2009)
6. Camenisch, J., Dubovitskaya, M., Neven, G.: Unlinkable priced oblivious transfer with rechargeable wallets. In: International Conference on Financial Cryptography and Data Security. pp. 66–81. Springer (2010)
7. Camenisch, J., Dubovitskaya, M., Neven, G., Zaverucha, G.M.: Oblivious transfer with hidden access control policies. In: International Workshop on Public Key Cryptography. pp. 192–209. Springer (2011)
8. Camenisch, J., Dubovitskaya, M., Rial, A.: Concise UC zero-knowledge proofs for oblivious updatable databases , `http://hdl.handle.net/10993/39423`
9. Camenisch, J., Dubovitskaya, M., Rial, A.: Uc commitments for modular protocol design and applications to revocation and attribute tokens. In: Annual International Cryptology Conference. pp. 208–239. Springer (2016)
10. Camenisch, J., Hohenberger, S., Lysyanskaya, A.: Compact e-cash. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 302–321. Springer (2005)
11. Camenisch, J., Neven, G., et al.: Simulatable adaptive oblivious transfer. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 573–590. Springer (2007)

12. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science. pp. 136–145. IEEE (2001)
13. Canetti, R., Lindell, Y., Ostrovsky, R., Sahai, A.: Universally composable two-party and multi-party secure computation. In: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing. pp. 494–503. ACM (2002)
14. Canetti, R., Rabin, T.: Universal composition with joint state. In: Annual International Cryptology Conference. pp. 265–281. Springer (2003)
15. Catalano, D., Fiore, D.: Vector commitments and their applications. In: International Workshop on Public Key Cryptography. pp. 55–72. Springer (2013)
16. Coull, S., Green, M., Hohenberger, S.: Controlling access to an oblivious database using stateful anonymous credentials. In: International Workshop on Public Key Cryptography. pp. 501–520. Springer (2009)
17. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The second-generation onion router. Tech. rep., Naval Research Lab Washington DC (2004)
18. Henry, R., Olumofin, F., Goldberg, I.: Practical pir for electronic commerce. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 677–690. ACM (2011)
19. Libert, B., Ling, S., Mouhartem, F., Nguyen, K., Wang, H.: Adaptive oblivious transfer with access control from lattice assumptions. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 533–563. Springer (2017)
20. Libert, B., Yung, M.: Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In: Theory of Cryptography Conference. pp. 499–517. Springer (2010)
21. Rial, A.: UC updatable non-hiding committed database with efficient zero-knowledge proofs , `http://hdl.handle.net/10993/39421`
22. Rial, A., Balasch, J., Preneel, B.: A privacy-preserving buyer–seller watermarking protocol based on priced oblivious transfer. IEEE Transactions on Information Forensics and Security **6**(1), 202–212 (2011)
23. Rial, A., Kohlweiss, M., Preneel, B.: Universally composable adaptive priced oblivious transfer. In: International Conference on Pairing-Based Cryptography. pp. 231–247. Springer (2009)
24. Rial, A., Preneel, B.: Optimistic fair priced oblivious transfer. In: International Conference on Cryptology in Africa. pp. 131–147. Springer (2010)
25. Zhang, Y., Au, M.H., Wong, D.S., Huang, Q., Mamoulis, N., Cheung, D.W., Yiu, S.M.: Oblivious transfer with access control: realizing disjunction without duplication. In: International Conference on Pairing-Based Cryptography. pp. 96–115. Springer (2010)

A Full Description of the Building Blocks of Our Construction

A.1 Ideal Functionality $\mathcal{F}_{\mathrm{AUT}}$

Our protocol uses the functionality $\mathcal{F}_{\mathrm{AUT}}$ for an authenticated channel in [12]. $\mathcal{F}_{\mathrm{AUT}}$ interacts with a sender $\mathcal{T}$ and a receiver $\mathcal{R}$, and consists of one interface $\mathsf{aut.send}$. $\mathcal{T}$ uses the $\mathsf{aut.send}$ interface to send a message m to $\mathcal{F}_{\mathrm{AUT}}$. $\mathcal{F}_{\mathrm{AUT}}$ leaks m to the simulator $\mathcal{S}$ and, after receiving a response from $\mathcal{S}$, $\mathcal{F}_{\mathrm{AUT}}$ sends m to $\mathcal{R}$. $\mathcal{S}$ cannot modify m. The session identifier sid contains the identities of $\mathcal{T}$ and $\mathcal{R}$.

Description of $\mathcal{F}_{\mathrm{AUT}}$. $\mathcal{F}_{\mathrm{AUT}}$ is parameterized by a message space $\mathcal{M}$.

1. On input $(\mathsf{aut.send.ini}, sid, m)$ from a party $\mathcal{T}$:
   - Abort if $sid \neq (\mathcal{T}, \mathcal{R}, sid')$ or if $m \notin \mathcal{M}$.
   - Create a fresh qid and store $(qid, \mathcal{R}, m)$.
   - Send $(\mathsf{aut.send.sim}, sid, qid, m)$ to $\mathcal{S}$.

S. On input $(\mathsf{aut.send.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid, \mathcal{R}, m)$ is not stored.
   - Delete the record $(qid, \mathcal{R}, m)$.
   - Send $(\mathsf{aut.send.end}, sid, m)$ to $\mathcal{R}$.

A.2 Ideal Functionality $\mathcal{F}_{\mathrm{SMT}}$

Our protocol uses the functionality $\mathcal{F}_{\mathrm{SMT}}$ for secure message transmission described in [12]. $\mathcal{F}_{\mathrm{SMT}}$ interacts with a sender $\mathcal{T}$ and a receiver $\mathcal{R}$, and consists of one interface $\mathsf{smt.send}$. $\mathcal{T}$ uses the $\mathsf{smt.send}$ interface to send a message m to $\mathcal{F}_{\mathrm{SMT}}$. $\mathcal{F}_{\mathrm{SMT}}$ leaks $l(m)$, where $l : \mathcal{M} \rightarrow \mathbb{N}$ is a function that leaks the message length, to the simulator $\mathcal{S}$. After receiving a response from $\mathcal{S}$, $\mathcal{F}_{\mathrm{SMT}}$ sends m to $\mathcal{R}$. $\mathcal{S}$ cannot modify m. The session identifier sid contains the identities of $\mathcal{T}$ and $\mathcal{R}$.

Description of $\mathcal{F}_{\mathrm{SMT}}$. $\mathcal{F}_{\mathrm{SMT}}$ is parameterized by a message space $\mathcal{M}$ and by a leakage function $l : \mathcal{M} \rightarrow \mathbb{N}$, which leaks the message length.

1. On input $(\mathsf{smt.send.ini}, sid, m)$ from a party $\mathcal{T}$:
   - Abort if $sid \neq (\mathcal{T}, \mathcal{R}, sid')$ or if $m \notin \mathcal{M}$.
   - Create a fresh qid and store $(qid, \mathcal{R}, m)$.
   - Send $(\mathsf{smt.send.sim}, sid, qid, l(m))$ to $\mathcal{S}$.

S. On input $(\mathsf{smt.send.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid, \mathcal{R}, m)$ is not stored.
   - Delete the record $(qid, \mathcal{R}, m)$.
   - Send $(\mathsf{smt.send.end}, sid, m)$ to $\mathcal{R}$.

A.3 Ideal Functionality $\mathcal{F}_{\mathrm{NIC}}$ for Non-Interactive Commitments

Our protocol uses the functionality $\mathcal{F}_{\mathrm{NIC}}$ for non-interactive commitments in [9]. $\mathcal{F}_{\mathrm{NIC}}$ interacts with parties $\mathcal{P}_i$ and consists of the following interfaces:

1. Any party $\mathcal{P}_i$ uses the com.setup interface to set up the functionality.
2. Any party $\mathcal{P}_i$ uses the com.commit interface to send a message cm and obtain a commitment $ccom$ and an opening $copen$. A commitment $ccom = (ccom'$, $cparcom, \mathsf{COM.Verify})$, where $ccom'$ is the commitment, $cparcom$ are the public parameters, and COM.Verify is the verification algorithm.
3. Any party $\mathcal{P}_i$ uses the com.validate interface to send a commitment $ccom$ in order to check that $ccom$ contains the correct public parameters and verification algorithm.
4. Any party $\mathcal{P}_i$ uses the com.verify interface to send $(ccom, cm, copen)$ in order to verify that $ccom$ is a commitment to the message cm with the opening $copen$.

$\mathcal{F}_{\mathrm{NIC}}$ can be realized by a perfectly hiding commitment scheme, such as Pedersen commitments [9]. In [9], a method is described to use $\mathcal{F}_{\mathrm{NIC}}$ in order to ensure that a party sends the same input cm to several ideal functionalities. For this purpose, the party first uses com.commit to get a commitment $ccom$ to cm with opening $copen$. Then the party sends $(ccom, cm, copen)$ as input to each of the functionalities, and each functionality runs COM.Verify to verify the commitment. Finally, other parties in the protocol receive the commitment $ccom$ from each of the functionalities and use the com.validate interface to validate $ccom$. Then, if $ccom$ received from all the functionalities is the same, the binding property provided by $\mathcal{F}_{\mathrm{NIC}}$ ensures that all the functionalities received the same input cm. When using $\mathcal{F}_{\mathrm{NIC}}$, it is needed to work in the $\mathcal{F}_{\mathrm{NIC}}||\mathcal{S}_{\mathrm{NIC}}$-hybrid model, where $\mathcal{S}_{\mathrm{NIC}}$ is any simulator for a construction that realizes $\mathcal{F}_{\mathrm{NIC}}$.

Description of $\mathcal{F}_{\mathrm{NIC}}$. COM.TrapCom, COM.TrapOpen and COM.Verify are ppt algorithms.

1. On input $(\mathsf{com.setup.ini}, sid)$ from a party $\mathcal{P}_i$:
   - If $(sid, cparcom, \mathsf{COM.TrapCom}, \mathsf{COM.TrapOpen}, \mathsf{COM.Verify}, ctdcom)$ is already stored, include $\mathcal{P}_i$ in the set $\mathbb{P}$, and send $(\mathsf{com.setup.end}, sid, OK)$ as a public delayed output to $\mathcal{P}_i$.
   - Otherwise proceed to generate a random qid, store $(qid, \mathcal{P}_i)$ and send the message $(\mathsf{com.setup.req}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{com.setup.alg}, sid, qid, m)$ from $\mathcal{S}$:
   - Abort if no pair $(qid, \mathcal{P}_i)$ for some $\mathcal{P}_i$ is stored.
   - Delete record $(qid, \mathcal{P}_i)$.
   - If $(sid, cparcom, \mathsf{COM.TrapCom}, \mathsf{COM.TrapOpen}, \mathsf{COM.Verify}, ctdcom)$ is already stored, include $\mathcal{P}_i$ in the set $\mathbb{P}$ and send $(\mathsf{com.setup.end}, sid, OK)$ to $\mathcal{P}_i$.
   - Otherwise proceed as follows.
     - m is $(cparcom, \mathsf{COM.TrapCom}, \mathsf{COM.TrapOpen}, \mathsf{COM.Verify}, ctdcom)$.
     - Initialize both an empty table Tbl_{com} and an empty set $\mathbb{P}$, and store $(sid$, $cparcom, \mathsf{COM.TrapCom}, \mathsf{COM.TrapOpen}, \mathsf{COM.Verify}, ctdcom)$.

    - Include $\mathcal{P}_i$ in the set $\mathbb{P}$ and send $(\mathsf{com.setup.end}, sid, OK)$ to $\mathcal{P}_i$.

2. On input $(\mathsf{com.validate.ini}, sid, ccom)$ from any party $\mathcal{P}_i$:
   - Abort if $\mathcal{P}_i \notin \mathbb{P}$.
   - Parse $ccom$ as $(ccom', cparcom', \mathsf{COM.Verify}')$.
   - Set $v \leftarrow 1$ if $cparcom' = cparcom$ and $\mathsf{COM.Verify}' = \mathsf{COM.Verify}$. Otherwise, set $v \leftarrow 0$.
   - Send $(\mathsf{com.validate.end}, sid, v)$ to $\mathcal{P}_i$.
3. On input $(\mathsf{com.commit.ini}, sid, cm)$ from any party $\mathcal{P}_i$:
   - Abort if $\mathcal{P}_i \notin \mathbb{P}$ or if $cm \notin \mathcal{M}$, where $\mathcal{M}$ is defined in $cparcom$.
   - Compute $(ccom, cinfo) \leftarrow \mathsf{COM.TrapCom}(sid, cparcom, ctdcom)$.
   - Abort if there is an entry $[ccom, cm', copen', 1]$ in Tbl_{com} such that $cm \neq cm'$.
   - Run $copen \leftarrow \mathsf{COM.TrapOpen}(sid, cm, cinfo)$.
   - Abort if $1 \neq \mathsf{COM.Verify}(sid, cparcom, ccom, cm, copen)$.
   - Append $[ccom, cm, copen, 1]$ to Tbl_{com}.
   - Set $ccom \leftarrow (ccom, cparcom, \mathsf{COM.Verify})$.
   - Send $(\mathsf{com.commit.end}, sid, ccom, copen)$ to $\mathcal{P}_i$.
4. On input $(\mathsf{com.verify.ini}, sid, ccom, cm, copen)$ from any party $\mathcal{P}_i$:
   - Abort if $\mathcal{P}_i \notin \mathbb{P}$ or if $cm \notin \mathcal{M}$ or if $copen \notin \mathcal{R}$, where $\mathcal{M}$ and $\mathcal{R}$ are defined in $cparcom$.
   - Parse $ccom$ as the tuple $(ccom', cparcom', \mathsf{COM.Verify}')$. Abort if the parameters $cparcom' \neq cparcom$ or $\mathsf{COM.Verify}' \neq \mathsf{COM.Verify}$.
   - If there is an entry $[ccom', cm, copen, u]$ in Tbl_{com}, set $v \leftarrow u$.
   - Else, proceed as follows:
     - If there is an entry $[ccom', cm', copen', 1]$ in Tbl_{com} such that $cm \neq cm'$, set $v \leftarrow 0$.
     - Else, proceed as follows:
       * Set $v \leftarrow \mathsf{COM.Verify}(sid, cparcom, ccom', cm, copen)$.
       * Append $[ccom', cm, copen, v]$ to Tbl_{com}.
   - Send $(\mathsf{com.verify.end}, sid, v)$ to $\mathcal{P}_i$.

A.4 Ideal Functionality $\mathcal{F}_{\mathrm{ZK}}^{R}$ for Zero-Knowledge

Let R be a polynomial time computable binary relation. For tuples $(wit, ins) \in R$ we call wit the witness and ins the instance. Our protocol uses the ideal functionality $\mathcal{F}_{\mathrm{ZK}}^{R}$ for zero-knowledge in [12]. $\mathcal{F}_{\mathrm{ZK}}^{R}$ is parameterized by a description of a relation R, runs with a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, and consists of one interface $\mathsf{zk.prove}$. $\mathcal{P}$ uses $\mathsf{zk.prove}$ to send a witness wit and an instance ins to $\mathcal{F}_{\mathrm{ZK}}^{R}$. $\mathcal{F}_{\mathrm{ZK}}^{R}$ checks whether $(wit, ins) \in R$, and, in that case, sends the instance ins to $\mathcal{V}$. The simulator $\mathcal{S}$ learns ins but not wit. In our POT protocol, we use relations that include commitments as part of the instance, while the committed value and the opening are part of the witness. The relation uses the verification algorithm of the commitment scheme to check correctness of the commitment. This allows us to use the method described in [9] to ensure that an input $\mathcal{F}_{\mathrm{ZK}}^{R}$ is equal to the input of other functionalities in our protocol.

Description of $\mathcal{F}_{\mathrm{ZK}}^{R}$. $\mathcal{F}_{\mathrm{ZK}}^{R}$ is parameterized by a description of a relation R. $\mathcal{F}_{\mathrm{ZK}}^{R}$ interacts with a prover $\mathcal{P}$ and a verifier $\mathcal{V}$.

1. On input ($\mathsf{zk.prove.ini}$, sid, wit, ins) from $\mathcal{P}$:
   - Abort if $sid \neq (\mathcal{P}, \mathcal{V}, sid')$ or if $(wit, ins) \notin R$.
   - Create a fresh qid and store (qid, ins).
   - Send ($\mathsf{zk.prove.sim}$, sid, qid, ins) to $\mathcal{S}$.

S. On input ($\mathsf{zk.prove.rep}$, sid, qid) from $\mathcal{S}$:
   - Abort if (qid, ins) is not stored.
   - Parse sid as $(\mathcal{P}, \mathcal{V}, sid')$.
   - Delete the record (qid, ins).
   - Send ($\mathsf{zk.prove.end}$, sid, ins) to $\mathcal{V}$.

A.5 Ideal Functionality $\mathcal{F}_{\mathrm{OT}}$ for Oblivious Transfer

Our protocol uses the ideal functionality $\mathcal{F}_{\mathrm{OT}}$ for oblivious transfer. $\mathcal{F}_{\mathrm{OT}}$ interacts with a sender $\mathcal{E}$ and a receiver $\mathcal{R}$, and consists of three interfaces $\mathsf{ot.init}$, $\mathsf{ot.request}$ and $\mathsf{ot.transfer}$.

1. $\mathcal{E}$ uses the $\mathsf{ot.init}$ interface to send the messages $\langle m_n \rangle_{n=1}^{N}$ to $\mathcal{F}_{\mathrm{OT}}$. $\mathcal{F}_{\mathrm{OT}}$ stores $\langle m_n \rangle_{n=1}^{N}$ and sends the number N of messages to $\mathcal{R}$. The simulator $\mathcal{S}$ also learns N.
2. $\mathcal{R}$ uses the $\mathsf{ot.request}$ interface to send an index $\sigma \in [1, N]$, a commitment $ccom_\sigma$ and an opening $copen_\sigma$ to $\mathcal{F}_{\mathrm{OT}}$. $\mathcal{F}_{\mathrm{OT}}$ parses the commitment $ccom_\sigma$ as ($cparcom$, com_σ, $\mathsf{COM.Verify}$) and verifies the commitment by running $\mathsf{COM.Verify}$. $\mathcal{F}_{\mathrm{OT}}$ stores $[\sigma, ccom_\sigma]$ and sends $ccom_\sigma$ to $\mathcal{E}$.
3. $\mathcal{E}$ uses the $\mathsf{ot.transfer}$ interface to send a commitment $ccom_\sigma$ to $\mathcal{F}_{\mathrm{OT}}$. If a tuple $[\sigma, ccom_\sigma]$ is stored, $\mathcal{F}_{\mathrm{OT}}$ sends the message m_σ to $\mathcal{R}$.

$\mathcal{F}_{\mathrm{OT}}$ is similar to existing functionalities for OT [11], except that it receives a commitment $ccom_\sigma$ to the index σ and an opening $copen_\sigma$ for that commitment. In addition, the transfer phase is split up into two interfaces $\mathsf{ot.request}$ and $\mathsf{ot.transfer}$, so that $\mathcal{E}$ receives $ccom_\sigma$ in the request phase. These changes are needed to use in our POT protocol the method in [9] to ensure that, when purchasing an item, the buyer sends the same index σ to $\mathcal{F}_{\mathrm{OT}}$ and to other functionalities. It is generally easy to modify existing UC OT protocols so that they realize our functionality $\mathcal{F}_{\mathrm{OT}}$.

Description of $\mathcal{F}_{\mathrm{OT}}$. Functionality $\mathcal{F}_{\mathrm{OT}}$ runs with a sender $\mathcal{E}$ and a receiver $\mathcal{R}$, and is parameterised with a maximum number of messages $\mathcal{N}_{max}$ and a message space $\mathcal{M}$.

1. On input ($\mathsf{ot.init.ini}$, sid, $\langle m_n \rangle_{n=1}^{N}$) from $\mathcal{E}$:
   (a) Abort if $sid \notin (\mathcal{E}, \mathcal{R}, sid')$, or if $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$ is already stored, or if $N > \mathcal{N}_{max}$.
   (b) Abort if for $n = 1$ to N, $m_n \notin \mathcal{M}$.
   (c) Store $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$.
   (d) Send ($\mathsf{ot.init.sim}$, sid, N) to S.

S. On input ($\mathsf{ot.init.rep}$, sid) from S:

(a) Abort if $(sid, \langle m_n \rangle_{n=1}^{N}, 0)$ is not stored, or if $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ is already stored.
(b) Store $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ and initialize an empty table Tbl_{ot}.
(c) Send $(\mathsf{ot.init.end}, sid, N)$ to $\mathcal{R}$.

2. On input $(\mathsf{ot.request.ini}, sid, \sigma, ccom_\sigma, copen_\sigma)$ from $\mathcal{R}$:
   (a) Abort if $(sid, \langle m_n \rangle_{n=1}^{N}, 1)$ is not stored.
   (b) Abort if $\sigma \notin [1, N]$.
   (c) Parse $ccom_\sigma$ as $(cparcom, com_\sigma, \mathsf{COM.Verify})$.
   (d) Abort if $\mathsf{COM.Verify}$ is not a ppt algorithm, or if $1 \neq \mathsf{COM.Verify}(cparcom, com_\sigma, copen_\sigma, \sigma)$.
   (e) Create a fresh qid and store $(qid, \sigma, ccom_\sigma)$.
   (f) Send $(\mathsf{ot.request.sim}, sid, qid, ccom_\sigma)$ to S.

S. On input $(\mathsf{ot.request.rep}, sid, qid)$ from S:
   (a) Abort if $(qid, \sigma, ccom_\sigma)$ is not stored.
   (b) Append $[\sigma, ccom_\sigma]$ to Tbl_{ot}.
   (c) Delete the record $(qid, \sigma, ccom_\sigma)$.
   (d) Send $(\mathsf{ot.request.end}, sid, ccom_\sigma)$ to $\mathcal{E}$.

3. On input $(\mathsf{ot.transfer.ini}, sid, ccom_\sigma)$ from $\mathcal{E}$:
   (a) Abort if there is no entry $[\sigma, ccom_\sigma]$ in Tbl_{ot}.
   (b) Create a fresh qid and store $(qid, ccom_\sigma)$.
   (c) Send $(\mathsf{ot.transfer.sim}, sid, qid)$ to $\mathcal{S}$.

S. On input $(\mathsf{ot.transfer.rep}, sid, qid, b)$, if $\mathcal{E}$ is corrupt, or $(\mathsf{ot.transfer.rep}, sid, qid)$, if $\mathcal{E}$ is honest, from S:
   (a) Abort if $(qid, ccom_\sigma)$ is not stored.
   (b) If $\mathcal{E}$ is corrupt and $b = 0$, set $v \leftarrow \perp$.
   (c) Else, set $v \leftarrow m_\sigma$.
   (d) Delete the record $(qid, ccom_\sigma)$.
   (e) Send $(\mathsf{ot.transfer.end}, sid, v)$ to $\mathcal{R}$.

A.6 Ideal Functionality $\mathcal{F}_{\text{CD}}$ for a Committed Database

Our protocol uses the ideal functionality $\mathcal{F}_{\text{CD}}$ for a committed database in [8]. $\mathcal{F}_{\text{CD}}$ interacts with a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, and consists of three interfaces $\mathsf{cd.setup}$, $\mathsf{cd.read}$ and $\mathsf{cd.write}$.

1. $\mathcal{V}$ uses the $\mathsf{cd.setup}$ interface to initialize Tbl_{cd}. $\mathcal{F}_{\text{CD}}$ stores Tbl_{cd} and sends Tbl_{cd} to $\mathcal{P}$ and to the simulator $\mathcal{S}$.
2. $\mathcal{P}$ uses $\mathsf{cd.read}$ to send a position i and a value v_r to $\mathcal{F}_{\text{CD}}$, along with commitments and openings $(ccom_i, copen_i)$ and $(ccom_r, copen_r)$ to the position and value respectively. $\mathcal{F}_{\text{CD}}$ verifies the commitments and checks that there is an entry $[i, v_r]$ in the table Tbl_{cd}. In that case, $\mathcal{F}_{\text{CD}}$ sends $ccom_i$ and $ccom_r$ to $\mathcal{V}$. $\mathcal{S}$ also learns $ccom_i$ and $ccom_r$.
3. $\mathcal{P}$ uses $\mathsf{cd.write}$ to send a position i and a value v_w to $\mathcal{F}_{\text{CD}}$, along with commitments and openings $(ccom_i, copen_i)$ and $(ccom_w, copen_w)$ to the position and value respectively. $\mathcal{F}_{\text{CD}}$ verifies the commitments and then updates Tbl_{cd} to store v_w at position i. $\mathcal{F}_{\text{CD}}$ sends $ccom_i$ and $ccom_w$ to $\mathcal{V}$. $\mathcal{S}$ also learns $ccom_i$ and $ccom_w$.

Basically, $\mathcal{F}_{\mathrm{CD}}$ allows a prover $\mathcal{P}$ to prove to a verifier $\mathcal{V}$ that two commitments $ccom_i$ and $ccom_r$ commit to a position and value that are read from a table, and that two commitments $ccom_i$ and $ccom_w$ commit to a position and value that are written into the table. In [8], an efficient construction for $\mathcal{F}_{\mathrm{CD}}$ based on hiding vector commitments [20,15] is proposed. In our POT protocol, $\mathcal{F}_{\mathrm{CD}}$ is used to store and update the deposit of the buyer and the counters of the number of purchases for each of the item categories.

Description of $\mathcal{F}_{\mathrm{CD}}$. Functionality $\mathcal{F}_{\mathrm{CD}}$ is parameterized by a universe of values U_v and by a maximum table size N_{max}. $\mathcal{F}_{\mathrm{CD}}$ interacts with a prover $\mathcal{P}$ and a verifier $\mathcal{V}$.

1. On input $(\mathsf{cd.setup.ini}, sid, \mathsf{Tbl}_{cd})$ from $\mathcal{V}$:
   - Abort if $sid \notin (\mathcal{P}, \mathcal{V}, sid')$ or if (sid, Tbl_{cd}) is already stored.
   - Abort if Tbl_{cd} does not consist of entries of the form $[i, v]$, or if the number of entries in Tbl_{cd} is not N_{max}.
   - Abort if for $i = 1$ to N_{max}, $v \notin U_v$ for any entry $[i, v]$ in Tbl_{cd}.
   - Initialize a counter $cv \leftarrow 0$ for the verifier and store (sid, cv) and (sid, Tbl_{cd}).
   - Send $(\mathsf{cd.setup.sim}, sid, \mathsf{Tbl}_{cd})$ to $\mathcal{S}$.

S. On input $(\mathsf{cd.setup.rep}, sid)$ from $\mathcal{S}$:
   - Abort if (sid, Tbl_{cd}) is not stored, or if $(sid, \mathsf{Tbl}_{cd}, cp)$ is already stored.
   - Initialize a counter $cp \leftarrow 0$ for the prover and store $(sid, \mathsf{Tbl}_{cd}, cp)$.
   - Send $(\mathsf{cd.setup.end}, sid, \mathsf{Tbl}_{cd})$ to $\mathcal{P}$.

2. On input $(\mathsf{cd.read.ini}, sid, ccom_i, i, copen_i, ccom_r, v_r, copen_r)$ from $\mathcal{P}$:
   - Abort if $(sid, \mathsf{Tbl}_{cd}, cp)$ is not stored.
   - Abort if $i \notin [1, N_{max}]$, or if $v_r \notin U_v$, or if $[i, v_r]$ is not stored in Tbl_{cd}.
   - Parse the commitment $ccom_i$ as $(ccom'_i, cparcom_i, \mathsf{COM.Verify}_i)$.
   - Parse the commitment $ccom_r$ as $(ccom'_r, cparcom_r, \mathsf{COM.Verify}_r)$.
   - Abort if $\mathsf{COM.Verify}_i$ or $\mathsf{COM.Verify}_r$ are not ppt algorithms.
   - Abort if $1 \neq \mathsf{COM.Verify}_i(cparcom_i, ccom_i, i, copen_i)$.
   - Abort if $1 \neq \mathsf{COM.Verify}_r(cparcom_r, ccom_r, v_r, copen_r)$.
   - Create a fresh qid and store $(qid, ccom_i, ccom_r, cp)$.
   - Send $(\mathsf{cd.read.sim}, sid, qid, ccom_i, ccom_r)$ to $\mathcal{S}$.

S. On input $(\mathsf{cd.read.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid, ccom_i, ccom_r, cp')$ is not stored.
   - Abort if $cp' \neq cv$, where cv is stored in (sid, cv).
   - Delete the record $(qid, ccom_i, ccom_r, cp')$.
   - Send $(\mathsf{cd.read.end}, sid, ccom_i, ccom_r)$ to $\mathcal{V}$.

3. On input $(\mathsf{cd.write.ini}, sid, ccom_i, i, copen_i, ccom_w, v_w, copen_w)$ from $\mathcal{P}$:
   - Abort if $(sid, \mathsf{Tbl}_{cd}, cp)$ is not stored.
   - Abort if $i \notin [1, N_{max}]$, or if $v_w \notin U_v$.
   - Parse the commitment $ccom_i$ as $(ccom'_i, cparcom_i, \mathsf{COM.Verify}_i)$.
   - Parse the commitment $ccom_w$ as $(ccom'_w, cparcom_w, \mathsf{COM.Verify}_w)$.
   - Abort if $\mathsf{COM.Verify}_i$ or $\mathsf{COM.Verify}_w$ are not ppt algorithms.
   - Abort if $1 \neq \mathsf{COM.Verify}_i(cparcom_i, ccom_i, i, copen_i)$.
   - Abort if $1 \neq \mathsf{COM.Verify}_w(cparcom_w, ccom_w, v_w, copen_w)$.
   - Increment the counter cp in $(sid, \mathsf{Tbl}_{cd}, cp)$ and store $[i, v_w]$ in Tbl_{cd}.

  - Create a fresh qid and store $(qid, ccom_i, ccom_w, cp)$.
  - Send (cd.write.sim, sid, qid, $ccom_i$, $ccom_w$) to $\mathcal{S}$.

S. On input (cd.write.rep, sid, qid) from $\mathcal{S}$:
  - Abort if $(qid, ccom_i, ccom_w, cp')$ is not stored.
  - Abort if $cp' \neq cv + 1$, where cv is stored in (sid, cv).
  - Increment the counter cv in (sid, cv).
  - Delete the record $(qid, ccom_i, ccom_w, cp')$.
  - Send (cd.write.end, sid, $ccom_i$, $ccom_w$) to $\mathcal{V}$.

A.7 Ideal Functionality $\mathcal{F}_{\mathrm{NHCD}}$ for a Non-Hiding Committed Database

Our protocol uses the ideal functionality $\mathcal{F}_{\mathrm{NHCD}}$ for a non-hiding committed database in [21]. $\mathcal{F}_{\mathrm{NHCD}}$ interacts with a party $\mathcal{P}_0$ and a party $\mathcal{P}_1$, and consists of three interfaces nhcd.setup, nhcd.prove and nhcd.write.

1. $\mathcal{P}_1$ uses nhcd.setup to send a table Tbl_{nhcd} with N entries of the form $[i, v]$ (for $i = 0$ to N) to $\mathcal{F}_{\mathrm{NHCD}}$. $\mathcal{F}_{\mathrm{NHCD}}$ stores Tbl_{nhcd} and sends Tbl_{nhcd} to $\mathcal{P}_0$. The simulator $\mathcal{S}$ also learns Tbl_{nhcd}.
2. $\mathcal{P}_b$ ($b \in [0, 1]$) uses nhcd.prove to send a position i and a value v_r to $\mathcal{F}_{\mathrm{NHCD}}$, along with commitments and openings along with commitments and openings $(ccom_i, copen_i)$ and $(ccom_r, copen_r)$ to the position and value respectively. $\mathcal{F}_{\mathrm{NHCD}}$ verifies the commitments and checks that there is an entry $[i, v_r]$ in the table Tbl_{nhcd}. In that case, $\mathcal{F}_{\mathrm{NHCD}}$ sends $ccom_i$ and $ccom_r$ to $\mathcal{P}_{1-b}$. The simulator $\mathcal{S}$ also learns $ccom_i$ and $ccom_r$.
3. $\mathcal{P}_1$ uses nhcd.write to send a position i and a value v_w to $\mathcal{F}_{\mathrm{NHCD}}$. $\mathcal{F}_{\mathrm{NHCD}}$ updates Tbl_{nhcd} to contain value v_w at position i and sends i and v_w to $\mathcal{P}_0$. The simulator $\mathcal{S}$ also learns i and v_w.

$\mathcal{F}_{\mathrm{NHCD}}$ is similar to the functionality $\mathcal{F}_{\mathrm{CD}}$ described above. The main difference is that the contents of the table Tbl_{nhcd} are known by both parties. For this reason, both parties can invoke the nhcd.prove interface to prove that two commitments $ccom_i$ and $ccom_r$ commit to a position and value stored in Tbl_{nhcd}. In addition, the interface nhcd.write reveals the updates to Tbl_{nhcd} made by $\mathcal{P}_1$ to $\mathcal{P}_0$. In [21], an efficient construction for $\mathcal{F}_{\mathrm{NHCD}}$ based on non-hiding vector commitments is proposed. In our POT protocol, $\mathcal{F}_{\mathrm{NHCD}}$ will be used by the seller, acting as $\mathcal{P}_1$, to store and update the prices of items. The buyer, acting as $\mathcal{P}_0$, uses the nhcd.prove interface to prove to the seller that the correct price for the item purchased is used.

Description of $\mathcal{F}_{\mathrm{NHCD}}$. Functionality $\mathcal{F}_{\mathrm{NHCD}}$ is parameterised by a universe of values U_v and by a maximum table size N. $\mathcal{F}_{\mathrm{NHCD}}$ interacts with a party $\mathcal{P}_0$ and a party $\mathcal{P}_1$. In the following, $b \in [0, 1]$.

1. On input (nhcd.setup.ini, sid, Tbl_{nhcd}) from $\mathcal{P}_1$:
   - Abort if $sid \notin (\mathcal{P}_0, \mathcal{P}_1, sid')$ or if $(sid, \mathsf{Tbl}'_{nhcd}, c_1)$ is already stored.
   - Abort if Tbl_{nhcd} does not consist of N entries of the form $[i, v]$.
   - Abort if for $i = 1$ to N, $v \notin U_v$ for any entry $[i, v]$ in Tbl_{nhcd}.
   - Initialize a counter $c_1 \leftarrow 0$ for $\mathcal{P}_1$ and store $(sid, \mathsf{Tbl}_{nhcd}, c_1)$.

  - Send $(\mathsf{nhcd.setup.sim}, sid, \mathsf{Tbl}_{nhcd})$ to $\mathcal{S}$.

S. On input $(\mathsf{nhcd.setup.rep}, sid)$ from $\mathcal{S}$:
  - Abort if $(sid, \mathsf{Tbl}_{nhcd}, c_1)$ is not stored, or if $(sid, \mathsf{Tbl}_{nhcd}, c_0)$ is stored.
  - Initialize a counter $c_0 \leftarrow 0$ for $\mathcal{P}_0$ and store $(sid, \mathsf{Tbl}_{nhcd}, c_0)$.
  - Send $(\mathsf{nhcd.setup.end}, sid, \mathsf{Tbl}_{nhcd})$ to $\mathcal{P}_0$.

2. On input $(\mathsf{nhcd.write.ini}, sid, i, v_w)$ from $\mathcal{P}_1$:
  - Abort if $(sid, \mathsf{Tbl}_{nhcd}, c_1)$ is not stored.
  - Abort if $i \notin [1, N]$, or if $v_w \notin U_v$.
  - Increment c_1 and update c_1 and the table entry $[i, v_w]$ in $(sid, \mathsf{Tbl}_{nhcd}, c_1)$.
  - Create a fresh qid and store (qid, i, v_w, c_1).
  - Send $(\mathsf{nhcd.write.sim}, sid, qid, i, v_w)$ to $\mathcal{S}$.

S. On input $(\mathsf{nhcd.write.rep}, sid, qid)$ from $\mathcal{S}$:
  - Abort if (qid, i, v_w, c_1') or $(sid, \mathsf{Tbl}_{nhcd}, c_0)$ are not stored, or if $c_1' \neq c_0 + 1$.
  - Increment c_0 and update c_0 and the table entry $[i, v_w]$ in $(sid, \mathsf{Tbl}_{nhcd}, c_0)$.
  - Delete the record (qid, i, v_w, c_1').
  - Send $(\mathsf{nhcd.write.end}, sid, i, v_w)$ to $\mathcal{P}_0$.

3. On input $(\mathsf{nhcd.prove.ini}, sid, ccom_i, i, copen_i, ccom_r, v_r, copen_r)$ from $\mathcal{P}_b$:
  - Abort if $(sid, \mathsf{Tbl}_{nhcd}, c_b)$ is not stored.
  - Abort if $i \notin [1, N]$, or if $v_r \notin U_v$, or if $[i, v_r]$ is not stored in Tbl_{nhcd}.
  - Parse $ccom_i$ as $(ccom_i', cparcom_i, \mathsf{COM.Verify}_i)$.
  - Parse $ccom_r$ as $(ccom_r', cparcom_r, \mathsf{COM.Verify}_r)$.
  - Abort if $cparcom_i \neq cparcom_r$, or if $\mathsf{COM.Verify}_i \neq \mathsf{COM.Verify}_r$, or if $\mathsf{COM.Verify}_i$ is not a ppt algorithm.
  - Abort if $1 \neq \mathsf{COM.Verify}_i(cparcom_i, ccom_i, i, copen_i)$.
  - Abort if $1 \neq \mathsf{COM.Verify}_r(cparcom_r, ccom_r, v_r, copen_r)$.
  - Create a fresh qid and store $(qid, ccom_i, ccom_r, \mathcal{P}_b, c_b)$.
  - Send $(\mathsf{nhcd.prove.sim}, sid, qid, ccom_i, ccom_r)$ to $\mathcal{S}$.

S. On input $(\mathsf{nhcd.prove.rep}, sid, qid)$ from $\mathcal{S}$:
  - Abort if $(qid, ccom_i, ccom_r, \mathcal{P}_b, c_b')$ or $(sid, \mathsf{Tbl}_{nhcd}, c_{1-b})$ are not stored, or if $c_b' \neq c_{1-b}$.
  - Delete the record $(qid, ccom_i, ccom_r, \mathcal{P}_b, c_b')$.
  - Send $(\mathsf{nhcd.prove.end}, sid, ccom_i, ccom_r)$ to $\mathcal{P}_{1-b}$.

B Full Description of Construction Π_{POTS}

Π_{POTS} uses $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$ and $\mathcal{F}_{\mathrm{NHCD}}$. Π_{POTS} is parameterised with a maximum number of messages $\mathcal{N}_{max}$, a message space $\mathcal{M}$, a maximum deposit value dep_{max}, a maximum price $\mathcal{P}_{max}$, and a universe of statistics Ψ consisting of ppt algorithms.

1. On input $(\mathsf{pot.init.ini}, sid, ep, \langle m_n \rangle_{n=1}^{N})$, $\mathcal{V}$ and $\mathcal{B}$ do the following:
  - $\mathcal{V}$ aborts if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$.
  - $\mathcal{V}$ aborts if (sid, ep', N) is already stored for $ep' = ep$, else stores (sid, ep, N).
  - $\mathcal{V}$ aborts if $N > \mathcal{N}_{max}$, or if for $n = 1$ to N, $m_n \notin \mathcal{M}$.

- If this is the first execution of this interface, $\mathcal{V}$ and $\mathcal{B}$ do the following:
  - $\mathcal{V}$ sets a table Tbl_{cd} of $\mathcal{N}_{max}$ entries where each entry is of the form $[i, 0]$ for $i = 0$ to $\mathcal{N}_{max}$.
  - $\mathcal{V}$ sends $(\mathsf{cd.setup.ini}, sid, \mathsf{Tbl}_{cd})$ to a new instance of $\mathcal{F}_{\mathrm{CD}}$.
  - $\mathcal{B}$ receives $(\mathsf{cd.setup.end}, sid, \mathsf{Tbl}_{cd})$ from $\mathcal{F}_{\mathrm{CD}}$.
  - If (sid, Tbl_{cd}) is already stored or if, for $i = 0$ to $\mathcal{N}_{max}$, there exists an entry $[i, v]$ in Tbl_{cd} such that $v \neq 0$, $\mathcal{B}$ aborts, else $\mathcal{B}$ stores (sid, Tbl_{cd}).
  - $\mathcal{B}$ sets $sid_{\mathrm{AUT}} \leftarrow (\mathcal{B}, \mathcal{V}, sid')$ and sends $(\mathsf{aut.send.ini}, sid_{\mathrm{AUT}}, \langle \mathsf{setup} \rangle)$ to $\mathcal{F}_{\mathrm{AUT}}$.
  - $\mathcal{V}$ receives $(\mathsf{aut.send.end}, sid_{\mathrm{AUT}}, \langle \mathsf{setup} \rangle)$ from $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{V}$ sets $sid_{\mathrm{OT}} \leftarrow (sid, ep)$.
- $\mathcal{V}$ sends $(\mathsf{ot.init.ini}, sid_{\mathrm{OT}}, \langle m_n \rangle_{n=1}^{N})$ to a new instance of $\mathcal{F}_{\mathrm{OT}}$.
- $\mathcal{B}$ receives $(\mathsf{ot.init.end}, sid_{\mathrm{OT}}, N)$ from the instance of $\mathcal{F}_{\mathrm{OT}}$.
- $\mathcal{B}$ takes ep from sid_{OT} and stores (sid, ep, N).
- $\mathcal{B}$ outputs $(\mathsf{pot.init.end}, sid, ep, N)$.

2. On input $(\mathsf{pot.setupprices.ini}, sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$, $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - $\mathcal{V}$ aborts if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$ or if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$ is already stored.
   - $\mathcal{V}$ aborts if, for $n = 1$ to $\mathcal{N}_{max}$, $p_n \notin (0, \mathcal{P}_{max}]$.
   - $\mathcal{V}$ stores $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$.
   - For $n = 1$ to $\mathcal{N}_{max}$, $\mathcal{V}$ sets a table Tbl_{nhcd} with entries $[n, p_n]$.
   - $\mathcal{V}$ sends $(\mathsf{nhcd.setup.ini}, sid, \mathsf{Tbl}_{nhcd})$ to a new instance of $\mathcal{F}_{\mathrm{NHCD}}$.
   - $\mathcal{B}$ receives $(\mathsf{nhcd.setup.end}, sid, \mathsf{Tbl}_{nhcd})$ from $\mathcal{F}_{\mathrm{NHCD}}$.
   - $\mathcal{B}$ parses Tbl_{nhcd} as $[n, p_n]$, for $n = 1$ to $\mathcal{N}_{max}$, and stores $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$.
   - $\mathcal{B}$ outputs $(\mathsf{pot.setupprices.end}, sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$.
3. On input $(\mathsf{pot.updateprice.ini}, sid, n, p)$, $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - $\mathcal{V}$ aborts if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$ is not stored.
   - $\mathcal{V}$ aborts if $n \notin [1, \mathcal{N}_{max}]$, or if $p \notin (0, \mathcal{P}_{max}]$.
   - $\mathcal{V}$ sends $(\mathsf{nhcd.write.ini}, sid, n, p)$ to $\mathcal{F}_{\mathrm{NHCD}}$.
   - $\mathcal{B}$ receives $(\mathsf{nhcd.write.end}, sid, n, p)$ from $\mathcal{F}_{\mathrm{NHCD}}$.
   - $\mathcal{B}$ sets $p_n \leftarrow p)$ and updates the stored tuple $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$.
   - $\mathcal{B}$ outputs $(\mathsf{pot.updateprice.end}, sid, n, p)$.
4. On input $(\mathsf{pot.deposit.ini}, sid, dep)$, $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - $\mathcal{B}$ aborts if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$.
   - $\mathcal{B}$ aborts if (sid, ep, N) is not stored for any ep.
   - $\mathcal{B}$ retrieves $[0, v]$ from Tbl_{cd} and sets $dep_1 \leftarrow v$.
   - $\mathcal{B}$ aborts if $dep_1 + dep \notin [0, dep_{max}]$.
   - If this is the first execution of the deposit interface, $\mathcal{B}$ does the following:
     - $\mathcal{B}$ sends $(\mathsf{com.setup.ini}, sid)$ to $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{B}$ receives $(\mathsf{com.setup.end}, sid, OK)$ from $\mathcal{F}_{\mathrm{NIC}}$.
   - If $(sid, ccom_{dep_2}, copen_{dep_2})$ is already stored, $\mathcal{B}$ sets $ccom_{dep_1} \leftarrow ccom_{dep_2}$ and $copen_{dep_1} \leftarrow copen_{dep_2}$. Otherwise, $\mathcal{B}$ does the following:
     - $\mathcal{B}$ sends $(\mathsf{com.commit.ini}, sid, dep_1)$ to $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, ccom_{dep_1}, copen_{dep_1})$ from $\mathcal{F}_{\mathrm{NIC}}$.
   - $\mathcal{B}$ sets $dep_2 \leftarrow dep_1 + dep$.
   - $\mathcal{B}$ sends $(\mathsf{com.commit.ini}, sid, dep)$ to $\mathcal{F}_{\mathrm{NIC}}$.

- $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, ccom_{dep}, copen_{dep})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{B}$ sends $(\mathsf{com.commit.ini}, sid, dep_2)$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, ccom_{dep_2}, copen_{dep_2})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{B}$ sets $wit_{dep} \leftarrow (dep, copen_{dep}, dep_1, copen_{dep_1}, dep_2, copen_{dep_2})$.
- $\mathcal{B}$ parses the commitment $ccom_{dep}$ as $(ccom'_{dep}, cparcom, \mathsf{COM.Verify})$, the commitment $ccom_{dep_1}$ as $(ccom'_{dep_1}, cparcom, \mathsf{COM.Verify})$, and the commitment $ccom_{dep_2}$ as $(ccom'_{dep_2}, cparcom, \mathsf{COM.Verify})$.
- $\mathcal{B}$ sets $ins_{dep} \leftarrow (cparcom, ccom'_{dep}, ccom'_{dep_1}, ccom'_{dep_2})$.
- $\mathcal{B}$ stores $(sid, wit_{dep}, ins_{dep}, \text{writedeposit})$.
- $\mathcal{B}$ sets $sid_{\mathrm{ZK}} \leftarrow (\mathcal{B}, \mathcal{V}, sid')$, and sends $(\mathsf{zk.prove.ini}, sid_{\mathrm{ZK}}, wit_{dep}, ins_{dep})$ to a new instance of $\mathcal{F}_{\mathrm{ZK}}^{R_{dep}}$. The R_{dep} is defined as follows.

$$\begin{aligned} R_{dep} =\{(wit_{dep}, ins_{dep}) :& \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{dep}, dep, copen_{dep}) \wedge \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{dep_1}, dep_1, copen_{dep_1}) \wedge \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{dep_2}, dep_2, copen_{dep_2}) \wedge \\ & dep_2 = dep + dep_1 \ \wedge dep_2 \in [0, dep_{max}]\} \end{aligned}$$

- $\mathcal{V}$ receives $(\mathsf{zk.prove.end}, sid_{\mathrm{ZK}}, ins_{dep})$ from $\mathcal{F}_{\mathrm{ZK}}^{R_{dep}}$.
- If this is the first execution of the deposit interface, $\mathcal{V}$ does the following:
  - $\mathcal{V}$ sends $(\mathsf{com.setup.ini}, sid)$ to $\mathcal{F}_{\mathrm{NIC}}$.
  - $\mathcal{V}$ receives $(\mathsf{com.setup.end}, sid, OK)$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ parses ins_{dep} as $(cparcom, ccom'_{dep}, ccom'_{dep_1}, ccom'_{dep_2})$.
- $\mathcal{V}$ sets the commitment $ccom_{dep} \leftarrow (ccom'_{dep}, cparcom, \mathsf{COM.Verify})$, the commitment $ccom_{dep_1} \leftarrow (ccom'_{dep_1}, cparcom, \mathsf{COM.Verify})$, and the commitment $ccom_{dep_2} \leftarrow (ccom'_{dep_2}, cparcom, \mathsf{COM.Verify})$.
- $\mathcal{V}$ aborts if $(sid, ccom'_{dep_2})$ is stored and $ccom'_{dep_2} \neq ccom_{dep_1}$.
- If $(sid, ccom'_{dep_2})$ is not stored, $\mathcal{V}$ does the following:
  - $\mathcal{V}$ sends $(\mathsf{com.validate.ini}, sid, ccom_{dep_1})$ to $\mathcal{F}_{\mathrm{NIC}}$.
  - $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_{dep_1})$ from $\mathcal{F}_{\mathrm{NIC}}$.
  - $\mathcal{V}$ aborts if $b_{dep_1} \neq 1$.
- $\mathcal{V}$ sends $(\mathsf{com.validate.ini}, sid, ccom_{dep_2})$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_{dep_2})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ sends $(\mathsf{com.validate.ini}, sid, ccom_{dep})$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_{dep})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ aborts if $b_{dep_2} = b_{dep} = 1$ does not hold.
- $\mathcal{V}$ stores (sid, ins_{dep}).
- $\mathcal{V}$ sets $sid'_{\mathrm{AUT}} \leftarrow (sid)$ and sends $(\mathsf{aut.send.ini}, sid'_{\mathrm{AUT}}, \langle \text{writedeposit} \rangle)$ to $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid'_{\mathrm{AUT}}, \langle \text{writedeposit} \rangle)$ from $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ aborts if $(sid, wit_{dep}, ins_{dep}, \text{writedeposit})$ is not stored.
- $\mathcal{B}$ parses ins_{dep} as $(cparcom, ccom_{dep}, ccom_{dep_1}, ccom_{dep_2})$.
- $\mathcal{B}$ parses wit_{dep} as $(dep, copen_{dep}, dep_1, copen_{dep_1}, dep_2, copen_{dep_2})$.

- $\mathcal{B}$ deletes $(sid, wit_{dep}, ins_{dep}, \text{writedeposit})$ and stores $(sid, wit_{dep}, ins_{dep}, \text{revealdeposit})$.
- If this is the first execution of this interface, $\mathcal{B}$ does the following:
  - $\mathcal{B}$ sends $(\mathsf{com.commit.ini}, sid, 0)$ to $\mathcal{F}_{\text{NIC}}$.
  - $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, ccom_0, copen_0)$ from $\mathcal{F}_{\text{NIC}}$.
  - $\mathcal{B}$ stores $(sid, ccom_0, copen_0)$.
- $\mathcal{B}$ stores $(sid, ccom_{dep_2}, copen_{dep_2})$.
- $\mathcal{B}$ sends to $\mathcal{F}_{\text{CD}}$ the message $(\mathsf{cd.write.ini}, sid, ccom_0, 0, copen_0, ccom_{dep_2}, dep_2, copen_{dep_2})$.
- $\mathcal{V}$ receives $(\mathsf{cd.write.end}, sid, ccom_0, ccom_{dep_2})$ from $\mathcal{F}_{\text{CD}}$.
- $\mathcal{V}$ aborts if (sid, ins_{dep}) is not stored.
- $\mathcal{V}$ aborts if the commitment $ccom_{dep_2}$ in ins_{dep} is not the same as that received from $\mathcal{F}_{\text{CD}}$, or if $ccom_0$ stored in $(sid, ccom_0)$ is not the same as the commitment received from $\mathcal{F}_{\text{CD}}$.
- $\mathcal{V}$ sends the message $(\mathsf{aut.send.ini}, sid_{\text{AUT}}, \langle \text{revealdeposit} \rangle)$ to $\mathcal{F}_{\text{AUT}}$.
- $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid_{\text{AUT}}, \langle \text{revealdeposit} \rangle)$ from $\mathcal{F}_{\text{AUT}}$.
- $\mathcal{B}$ aborts if $(sid, wit_{dep}, ins_{dep}, \text{revealdeposit})$ is not stored.
- $\mathcal{B}$ deletes the record $(sid, wit_{dep}, ins_{dep}, \text{revealdeposit})$.
- $\mathcal{B}$ updates Tbl_{cd} with $[0, dep_2]$.
- $\mathcal{B}$ sets $sid_{\text{SMT}} \leftarrow (\mathcal{B}, \mathcal{V}, sid')$.
- If this is the first execution of this interface, $\mathcal{B}$ and $\mathcal{V}$ do the following:
  - $\mathcal{B}$ sends the following message $(\mathsf{smt.send.ini}, sid_{\text{SMT}}, \langle dep, copen_{dep}, 0, copen_0, dep_1, copen_{dep_1} \rangle)$ to functionality $\mathcal{F}_{\text{SMT}}$.
  - $\mathcal{V}$ receives from functionality $\mathcal{F}_{\text{SMT}}$ the following message $(\mathsf{smt.send.end}, sid_{\text{SMT}}, \langle dep, copen_{dep}, 0, copen_0, dep_1, copen_{dep_1} \rangle)$.
  - $\mathcal{V}$ sends $(\mathsf{com.verify.ini}, sid, ccom_0, 0, copen_0)$ to $\mathcal{F}_{\text{NIC}}$.
  - $\mathcal{V}$ receives $(\mathsf{com.verify.end}, sid, v_0)$ from $\mathcal{F}_{\text{NIC}}$ and aborts if $v_0 \neq 1$.
  - $\mathcal{V}$ sends $(\mathsf{com.verify.ini}, sid, ccom_{dep_1}, dep_1, copen_{dep_1})$ to $\mathcal{F}_{\text{NIC}}$.
  - $\mathcal{V}$ receives $(\mathsf{com.verify.end}, sid, v_{dep_1})$ from $\mathcal{F}_{\text{NIC}}$ and aborts if $v_{dep_1} \neq 1$.
  - $\mathcal{V}$ aborts if $dep_1 = 0$ does not hold.
- Otherwise, $\mathcal{B}$ and $\mathcal{V}$ do the following:
  - $\mathcal{B}$ sends the message $(\mathsf{smt.send.ini}, sid_{\text{SMT}}, \langle dep, copen_{dep} \rangle)$ to $\mathcal{F}_{\text{SMT}}$.
  - $\mathcal{V}$ receives $(\mathsf{smt.send.end}, sid_{\text{SMT}}, \langle dep, copen_{dep} \rangle)$ from $\mathcal{F}_{\text{SMT}}$.
- $\mathcal{V}$ sends $(\mathsf{com.verify.ini}, sid, ccom_{dep}, dep, copen_{dep})$ to $\mathcal{F}_{\text{NIC}}$.
- $\mathcal{V}$ receives $(\mathsf{com.verify.end}, sid, v)$ from $\mathcal{F}_{\text{NIC}}$ and aborts if $v \neq 1$.
- $\mathcal{V}$ aborts if $dep \notin [0, dep_{max}]$.
- $\mathcal{V}$ outputs $(\mathsf{pot.deposit.end}, sid, dep)$.

5. On input $(\mathsf{pot.transfer.ini}, sid, ep, \sigma)$, $\mathcal{V}$ and $\mathcal{B}$ do the following:
   - $\mathcal{B}$ aborts if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$.
   - $\mathcal{B}$ aborts if (sid, ep, N) is not stored, or if $(sid, \langle p_n \rangle_{n=1}^{\mathcal{N}_{max}})$ is not stored, or if $\sigma \notin [1, N]$.
   - $\mathcal{B}$ retrieves $[0, v]$ from Tbl_{cd}, and sets $dep_1 \leftarrow v$.
   - $\mathcal{B}$ aborts if $dep_1 < p_\sigma$.
   - $\mathcal{B}$ sets $ccom_{dep_1} \leftarrow ccom_{dep_2}$ and $copen_{dep_1} \leftarrow copen_{dep_2}$.

- $\mathcal{B}$ sets $dep_2 \leftarrow dep_1 - p_\sigma$.
- $\mathcal{B}$ sends $(\mathsf{com.commit.ini}, sid, dep_2)$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, ccom_{dep_2}, copen_{dep_2})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{B}$ sends $(\mathsf{com.commit.ini}, sid, \sigma)$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, ccom_\sigma, copen_\sigma)$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{B}$ sends $(\mathsf{com.commit.ini}, sid, p_\sigma)$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, ccom_{p_\sigma}, copen_{p_\sigma})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{B}$ sets $wit_{trans} \leftarrow (p_\sigma, copen_{p_\sigma}, dep_1, copen_{dep_1}, dep_2, copen_{dep_2})$.
- $\mathcal{B}$ parses the commitment $ccom_{p_\sigma}$ as $(ccom'_{p_\sigma}, cparcom, \mathsf{COM.Verify})$, the commitment $ccom_{dep_1}$ as $(ccom'_{dep_1}, cparcom, \mathsf{COM.Verify})$, and the commitment $ccom_{dep_2}$ as $(ccom'_{dep_2}, cparcom, \mathsf{COM.Verify})$.
- $\mathcal{B}$ sets $ins_{trans} \leftarrow (cparcom, ccom'_{p_\sigma}, ccom'_{dep_1}, ccom'_{dep_2})$.
- $\mathcal{B}$ stores $(sid, wit_{trans}, ins_{trans}, \text{readprice})$.
- $\mathcal{B}$ sets $sid_{\mathrm{ZK}} \leftarrow (\mathcal{B}, \mathcal{V}, sid')$, and sends the message $(\mathsf{zk.prove.ini}, sid_{\mathrm{ZK}}, wit_{trans}, ins_{trans})$ to a new instance of $\mathcal{F}_{\mathrm{ZK}}^{R_{trans}}$. The relation R_{trans} is defined as follows:

$$\begin{aligned} R_{trans} = \{ & (wit_{trans}, ins_{trans}) : \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{p_\sigma}, p_\sigma, copen_{p_\sigma}) \wedge \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{dep_1}, dep_1, copen_{dep_1}) \wedge \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{dep_2}, dep_2, copen_{dep_2}) \wedge \\ & dep_2 = dep_1 - p_\sigma \ \wedge \ dep_2 \in [0, dep_{max}]\} \end{aligned}$$

- $\mathcal{V}$ receives $(\mathsf{zk.prove.end}, sid_{\mathrm{ZK}}, ins_{trans})$ from $\mathcal{F}_{\mathrm{ZK}}^{R_{trans}}$.
- $\mathcal{V}$ parses ins_{trans} as $(cparcom, ccom'_{p_\sigma}, ccom'_{dep_1}, ccom'_{dep_2}, pk)$.
- $\mathcal{V}$ sets the commitment $ccom_{p_\sigma} \leftarrow (ccom'_{p_\sigma}, cparcom, \mathsf{COM.Verify})$, the commitment $ccom_{dep_1} \leftarrow (ccom'_{dep_1}, cparcom, \mathsf{COM.Verify})$, and the commitment $ccom_{dep_2} \leftarrow (ccom'_{dep_2}, cparcom, \mathsf{COM.Verify})$.
- $\mathcal{V}$ sends $(\mathsf{com.validate.ini}, sid, ccom_{p_\sigma})$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_{p_\sigma})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ aborts if $b_{p_\sigma} \neq 1$.
- $\mathcal{V}$ sends $(\mathsf{com.validate.ini}, sid, ccom_{dep_2})$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_{dep_2})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ aborts if $b_{dep_2} \neq 1$.
- $\mathcal{V}$ aborts if $ccom_{dep_2}$ stored in $(sid, ccom_{dep_2})$ is not the same as $ccom_{dep_1}$ in ins_{trans}.
- $\mathcal{V}$ stores (sid, ins_{trans}).
- $\mathcal{V}$ sets $sid'_{\mathrm{AUT}} \leftarrow (sid)$ and sends to functionality $\mathcal{F}_{\mathrm{AUT}}$ the following message $(\mathsf{aut.send.ini}, sid'_{\mathrm{AUT}}, \langle\text{readprice}\rangle)$.
- $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid_{\mathrm{AUT}}, \langle\text{readprice}\rangle)$ from $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ aborts if $(sid, wit_{trans}, ins_{trans}, \text{readprice})$ is not stored.
- $\mathcal{B}$ deletes the tuple $(sid, wit_{trans}, ins_{trans}, \text{readprice})$ and stores $(sid, wit_{trans}, ins_{trans}, \text{commitdeposit})$.
- $\mathcal{B}$ sends to functionality $\mathcal{F}_{\mathrm{NHCD}}$ the following message $(\mathsf{nhcd.prove.ini}, sid, ccom_\sigma, \sigma, copen_\sigma, ccom_{p_\sigma}, p_\sigma, copen_{p_\sigma})$.

– $\mathcal{B}$ receives $(\textsf{nhcd.prove.end}, sid, ccom_{\sigma}, ccom_{p_{\sigma}})$ from $\mathcal{F}_{\mathrm{NHCD}}$.
– $\mathcal{V}$ aborts if (sid, ins_{trans}) is not stored.
– $\mathcal{V}$ aborts if the $ccom_{p_{\sigma}}$ in ins_{trans} is not the same as that received from $\mathcal{F}_{\mathrm{NHCD}}$.
– $\mathcal{V}$ sends $(\textsf{com.validate.ini}, sid, ccom_{\sigma})$ to $\mathcal{F}_{\mathrm{NIC}}$.
– $\mathcal{V}$ receives $(\textsf{com.validate.end}, sid, b_{\sigma})$ from $\mathcal{F}_{\mathrm{NIC}}$.
– $\mathcal{V}$ aborts if $b_{\sigma} \neq 1$.
– $\mathcal{V}$ adds $ccom_{\sigma}$ to ins_{trans} and stores (sid, ins_{trans}).
– $\mathcal{V}$ sends the message $(\textsf{aut.send.ini}, sid'_{\mathrm{AUT}}, \langle \text{commitdeposit} \rangle)$ to $\mathcal{F}_{\mathrm{AUT}}$.
– $\mathcal{B}$ receives $(\textsf{aut.send.end}, sid'_{\mathrm{AUT}}, \langle \text{commitdeposit} \rangle)$ from $\mathcal{F}_{\mathrm{AUT}}$.
– $\mathcal{B}$ deletes the tuple $(sid, wit_{trans}, ins_{trans}, \text{commitdeposit})$ and stores $(sid, wit_{trans}, ins_{trans}, \text{commitcounter})$.
– $\mathcal{B}$ sends to $\mathcal{F}_{\mathrm{CD}}$ the message $(\textsf{cd.write.ini}, sid, ccom_0, 0, copen_0, ccom_{dep_2}, dep_2, copen_{dep_2})$.
– $\mathcal{V}$ receives $(\textsf{cd.write.end}, sid, ccom_0, ccom_{dep_2})$ from $\mathcal{F}_{\mathrm{CD}}$.
– $\mathcal{V}$ aborts if (sid, ins_{trans}) is not stored.
– $\mathcal{V}$ aborts if the $ccom_{dep_2}$ in ins_{trans} is not the same as that received from $\mathcal{F}_{\mathrm{CD}}$, or if $ccom_0$ received from $\mathcal{F}_{\mathrm{CD}}$ is not the same as $ccom_0$ stored by $\mathcal{V}$ during the first execution of the deposit interface.
– $\mathcal{V}$ sends the message $(\textsf{aut.send.ini}, sid'_{\mathrm{AUT}}, \langle \text{commitcounter} \rangle)$ to $\mathcal{F}_{\mathrm{AUT}}$.
– $\mathcal{B}$ receives $(\textsf{aut.send.end}, sid'_{\mathrm{AUT}}, \langle \text{commitcounter} \rangle)$ from $\mathcal{F}_{\mathrm{AUT}}$.
– $\mathcal{B}$ aborts if $(sid, wit_{trans}, ins_{trans}, \text{commitcounter})$ is not stored.
– $\mathcal{B}$ stores $(sid, ccom_{dep_2})$.
– $\mathcal{B}$ updates $\textsf{Tbl}_{cd}$ with $[0, dep_2]$.
– $\mathcal{B}$ retrieves $[\sigma, v]$ from $\textsf{Tbl}_{cd}$ and sets $count_1 \leftarrow v$.
– $\mathcal{B}$ sets $count_2 \leftarrow count_1 + 1$.
– $\mathcal{B}$ sends $(\textsf{com.commit.ini}, sid, count_1)$ to $\mathcal{F}_{\mathrm{NIC}}$.
– $\mathcal{B}$ receives $(\textsf{com.commit.end}, sid, ccom_{count_1}, copen_{count_1})$ from $\mathcal{F}_{\mathrm{NIC}}$.
– $\mathcal{B}$ sends $(\textsf{com.commit.ini}, sid, count_2)$ to $\mathcal{F}_{\mathrm{NIC}}$.
– $\mathcal{B}$ receives $(\textsf{com.commit.end}, sid, ccom_{count_2}, copen_{count_2})$ from $\mathcal{F}_{\mathrm{NIC}}$.
– $\mathcal{B}$ sets $wit_{count} \leftarrow (\sigma, copen_{\sigma}, count_1, copen_{count_1}, count_2, copen_{count_2})$.
– $\mathcal{B}$ parses the commitment $ccom_{\sigma}$ as $(ccom'_{\sigma}, cparcom, \textsf{COM.Verify})$, the commitment $ccom_{count_1}$ as $(ccom'_{count_1}, cparcom, \textsf{COM.Verify})$, and the commitment $ccom_{count_2}$ as $(ccom'_{count_2}, cparcom, \textsf{COM.Verify})$.
– $\mathcal{B}$ sets $ins_{count} \leftarrow (cparcom, ccom'_{\sigma}, ccom'_{count_1}, ccom'_{count_2})$.
– $\mathcal{B}$ deletes the tuple $(sid, wit_{trans}, ins_{trans}, \text{commitcounter})$ and stores the tuple $(sid, wit_{count}, ins_{count}, \text{readcounter})$.
– $\mathcal{B}$ sends $(\textsf{zk.prove.ini}, sid_{\mathrm{ZK}}, wit_{count}, ins_{count})$ to a new instance of $\mathcal{F}_{\mathrm{ZK}}^{R_{count}}$. The relation R_{count} is defined as follows:

$$\begin{aligned} R_{count} = \{ & (wit_{count}, ins_{count}) : \\ & 1 = \textsf{COM.Verify}(cparcom, ccom_{\sigma}, \sigma, copen_{\sigma}) \wedge \\ & 1 = \textsf{COM.Verify}(cparcom, ccom_{count_1}, count_1, copen_{count_1}) \wedge \\ & 1 = \textsf{COM.Verify}(cparcom, ccom_{count_2}, count_2, copen_{count_2}) \wedge \\ & count_2 = count_1 + 1 \} \end{aligned}$$

– $\mathcal{V}$ receives $(\textsf{zk.prove.end}, sid_{\mathrm{ZK}}, ins_{count})$ from $\mathcal{F}_{\mathrm{ZK}}^{R_{count}}$.

- $\mathcal{V}$ parses ins_{count} as $(cparcom, ccom'_{\sigma}, ccom'_{count_1}, ccom'_{count_2})$.
- $\mathcal{V}$ sets the commitment $ccom_{\sigma} \leftarrow (ccom'_{\sigma}, cparcom, \mathsf{COM.Verify})$, the commitment $ccom_{count_1} \leftarrow (ccom'_{count_1}, cparcom, \mathsf{COM.Verify})$, and the commitment $ccom_{count_2} \leftarrow (ccom'_{count_2}, cparcom, \mathsf{COM.Verify})$.
- $\mathcal{V}$ sends $(\mathsf{com.validate.ini}, sid, ccom_{count_1})$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_{count_1})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ aborts if $b_{count_1} \neq 1$.
- $\mathcal{V}$ sends $(\mathsf{com.validate.ini}, sid, ccom_{count_2})$ to $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_{count_2})$ from $\mathcal{F}_{\mathrm{NIC}}$.
- $\mathcal{V}$ aborts if $b_{count_2} \neq 1$.
- $\mathcal{V}$ aborts if the commitment $ccom_{\sigma}$ in ins_{trans} is not the same as the commitment $ccom_{\sigma}$ in ins_{count}.
- $\mathcal{V}$ stores (sid, ins_{count}).
- $\mathcal{V}$ sends the message $(\mathsf{aut.send.ini}, sid'_{\mathrm{AUT}}, \langle \text{readcounter} \rangle)$ to $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid'_{\mathrm{AUT}}, \langle \text{readcounter} \rangle)$ from $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ aborts if $(sid, wit_{count}, ins_{count}, \text{readcounter})$ is not stored.
- $\mathcal{B}$ parses ins_{count} as $(cparcom, ccom_{\sigma}, ccom_{count_1}, ccom_{count_2})$.
- $\mathcal{B}$ parses wit_{count} as $(\sigma, copen_{\sigma}, count_1, copen_{count_1}, count_2, copen_{count_2})$.
- $\mathcal{B}$ deletes the tuple $(sid, wit_{count}, ins_{count}, \text{readcounter})$ and stores the tuple $(sid, wit_{count}, ins_{count}, \text{writecounter})$.
- $\mathcal{B}$ sends the message $(\mathsf{cd.read.ini}, sid, ccom_{\sigma}, \sigma, copen_{\sigma}, ccom_{count_1}, count_1, copen_{count_1})$ to $\mathcal{F}_{\mathrm{CD}}$.
- $\mathcal{V}$ receives $(\mathsf{cd.read.end}, sid, ccom_{\sigma}, ccom_{count_1})$ from $\mathcal{F}_{\mathrm{CD}}$.
- $\mathcal{V}$ aborts if ins_{count} is not stored.
- $\mathcal{V}$ aborts if the commitments in ins_{count} are not the same as those received from $\mathcal{F}_{\mathrm{CD}}$.
- $\mathcal{V}$ sends the message $(\mathsf{aut.send.ini}, sid'_{\mathrm{AUT}}, \langle \text{writecounter} \rangle)$ to $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid'_{\mathrm{AUT}}, \langle \text{writecounter} \rangle)$ from $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ aborts if $(sid, wit_{count}, ins_{count}, \text{writecounter})$ is not stored.
- $\mathcal{B}$ deletes the record $(sid, wit_{count}, ins_{count}, \text{writecounter})$ and stores the tuple $(sid, wit_{count}, ins_{count}, \text{transfer})$.
- $\mathcal{B}$ sends to functionality $\mathcal{F}_{\mathrm{CD}}$the following message $(\mathsf{cd.write.ini}, sid, ccom_{\sigma}, \sigma, copen_{\sigma}, ccom_{count_2}, count_2, copen_{count_2})$.
- $\mathcal{V}$ receives $(\mathsf{cd.write.end}, sid, ccom_{\sigma}, ccom_{count_2})$ from $\mathcal{F}_{\mathrm{CD}}$.
- $\mathcal{V}$ aborts if ins_{count} is not stored.
- $\mathcal{V}$ aborts if the commitments in ins_{count} are not the same as those received from $\mathcal{F}_{\mathrm{CD}}$.
- $\mathcal{V}$ sends the message $(\mathsf{aut.send.ini}, sid'_{\mathrm{AUT}}, \langle \text{transfer} \rangle)$ to $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid'_{\mathrm{AUT}}, \langle \text{transfer} \rangle)$ from $\mathcal{F}_{\mathrm{AUT}}$.
- $\mathcal{B}$ aborts if $(sid, wit_{count}, ins_{count}, \text{transfer})$ is not stored.
- $\mathcal{B}$ updates Tbl_{cd} with $[\sigma, count_2]$.
- $\mathcal{B}$ sets $sid_{\mathrm{OT}} \leftarrow (sid, ep)$, and sends the message $(\mathsf{ot.request.ini}, sid_{\mathrm{OT}}, \sigma, ccom_{\sigma}, copen_{\sigma})$ to $\mathcal{F}_{\mathrm{OT}}$.
- $\mathcal{V}$ receives $(\mathsf{ot.request.end}, sid_{\mathrm{OT}}, ccom_{\sigma})$ from $\mathcal{F}_{\mathrm{OT}}$.
- $\mathcal{V}$ aborts if $ccom_{\sigma}$ received from $\mathcal{F}_{\mathrm{OT}}$ is not the same as that contained in ins_{count}.

- $\mathcal{V}$ sends the message $(\mathsf{ot.transfer.ini}, sid_{\mathrm{OT}}, ccom_{\sigma})$ to $\mathcal{F}_{\mathrm{OT}}$.
- $\mathcal{B}$ receives $(\mathsf{ot.transfer.end}, sid_{\mathrm{OT}}, m_{\sigma})$ from $\mathcal{F}_{\mathrm{OT}}$.
- $\mathcal{B}$ outputs $(\mathsf{pot.transfer.end}, sid, m_{\sigma})$.

6. On input $(\mathsf{pot.revealstatistic.ini}, sid, \mathsf{ST})$, $\mathcal{B}$ and $\mathcal{V}$ do the following:
   - $\mathcal{B}$ aborts if $sid \notin (\mathcal{V}, \mathcal{B}, sid')$.
   - $\mathcal{B}$ aborts if (sid, ep, N) is not stored for any ep.
   - $\mathcal{B}$ aborts if $\mathsf{ST} \notin \psi$.
   - $\mathcal{B}$ computes $result \leftarrow \mathsf{ST}(\mathsf{Tbl}_{cd})$.
   - For each entry $[i, v_i]$ in Tbl_{cd}, where v_i represents a value that was used by $\mathcal{B}$ to compute $result$, $\mathcal{B}$ and $\mathcal{V}$ do the following:
     - $\mathcal{B}$ sends the message $(\mathsf{com.commit.ini}, sid, i)$ to $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, ccom_i, copen_i)$.
     - $\mathcal{B}$ sends the message $(\mathsf{com.commit.ini}, sid, v_i)$ to $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{B}$ receives $(\mathsf{com.commit.end}, sid, ccom_{v_i}, copen_{v_i})$.
     - $\mathcal{B}$ stores $(sid, ccom_i, ccom_{v_i})$.
     - $\mathcal{B}$ sends $(\mathsf{cd.read.ini}, sid, ccom_i, i, copen_i, ccom_{v_i}, v_i, copen_{v_i})$ to $\mathcal{F}_{\mathrm{CD}}$.
     - $\mathcal{V}$ receives $(\mathsf{cd.read.end}, sid, ccom_i, ccom_{v_i})$ from $\mathcal{F}_{\mathrm{CD}}$.
     - $\mathcal{V}$ sends the message $(\mathsf{com.validate.ini}, sid, ccom_i)$ to $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_i)$ from $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{V}$ sends the message $(\mathsf{com.validate.ini}, sid, ccom_{v_i})$ to $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{V}$ receives $(\mathsf{com.validate.end}, sid, b_{v_i})$ from $\mathcal{F}_{\mathrm{NIC}}$.
     - $\mathcal{V}$ aborts if $b_i = b_{v_i} = 1$ does not hold.
     - $\mathcal{V}$ sets $sid_{\mathrm{AUT}} \leftarrow (sid)$ and sends $(\mathsf{aut.send.ini}, sid_{\mathrm{AUT}}, \langle OK, ccom_i, ccom_{v_i}\rangle)$ to $\mathcal{F}_{\mathrm{AUT}}$.
     - $\mathcal{B}$ receives $(\mathsf{aut.send.end}, sid_{\mathrm{AUT}}, \langle OK, ccom_i, ccom_{v_i}\rangle)$ from $\mathcal{F}_{\mathrm{AUT}}$.
     - $\mathcal{B}$ aborts if $(sid, ccom_i, ccom_{v_i})$ is not stored.
   - $\mathcal{B}$ sets $wit_{\mathsf{ST}} \leftarrow (\langle i, copen_i, v_i, copen_{v_i}\rangle_{\forall i})$.
   - $\mathcal{B}$ parses the commitment $ccom_i$ as $(ccom'_i, cparcom, \mathsf{COM.Verify})$ and the commitment $ccom_{v_i}$ as $(ccom'_{v_i}, cparcom, \mathsf{COM.Verify})$ for all i.
   - $\mathcal{B}$ sets $ins_{\mathsf{ST}} \leftarrow (result, cparcom, \langle ccom'_i, ccom'_{v_i}\rangle_{\forall i})$.
   - $\mathcal{B}$ sets $sid_{\mathrm{ZK}} \leftarrow (\mathcal{B}, \mathcal{V}, sid')$ and sends $(\mathsf{zk.prove.ini}, sid_{\mathrm{ZK}}, wit_{\mathsf{ST}}, ins_{\mathsf{ST}})$ to a new instance of $\mathcal{F}_{\mathrm{ZK}}^{R_{\mathsf{ST}}}$. The relation R_{ST} is defined as follows:

$$\begin{aligned} R_{\mathsf{ST}} =\{ & (wit_{\mathsf{ST}}, ins_{\mathsf{ST}}) : \\ & [\, \forall i \; 1 = \mathsf{COM.Verify}(cparcom, ccom_i, i, copen_i) \wedge \\ & 1 = \mathsf{COM.Verify}(cparcom, ccom_{v_i}, v_i, copen_{v_i}) \,] \wedge \\ & result = \mathsf{ST}(\langle i, v_i\rangle_{\forall i}) \;\} \end{aligned}$$

   - $\mathcal{V}$ receives $(\mathsf{zk.prove.end}, sid_{\mathrm{ZK}}, ins_{\mathsf{ST}})$ from $\mathcal{F}_{\mathrm{ZK}}^{R_{\mathsf{ST}}}$.
   - $\mathcal{V}$ aborts if the commitments received from $\mathcal{F}_{\mathrm{CD}}$ are not the same as those in ins_{ST}.
   - $\mathcal{V}$ outputs $(\mathsf{pot.revealstatistic.end}, sid, result, \mathsf{ST})$.

C Security Analysis of Construction $\mathit{\Pi}_{\mathrm{POTS}}$

To prove that our construction $\mathit{\Pi}_{\mathrm{POTS}}$ securely realizes the ideal functionality $\mathcal{F}_{\mathrm{POTS}}$, we have to show that for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish between whether it is interacting with $\mathcal{A}$ and the protocol in the real world or with $\mathcal{S}$ and $\mathcal{F}_{\mathrm{POTS}}$. The simulator thereby plays the role of all honest parties in the real world and interacts with $\mathcal{F}_{\mathrm{POTS}}$ for all corrupt parties in the ideal world.

Our simulator $\mathcal{S}$ runs copies of the functionalities $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$, and $\mathcal{F}_{\mathrm{NHCD}}$. When any of the copies of these functionalities abort, $\mathcal{S}$ implicitly forwards the abortion message to the adversary if the functionality sends the abortion message to a corrupt party.

In Section C.1, we analyze the security of construction Π_{POTS} when the buyer $\mathcal{B}$ is corrupt. In Section C.2, we analyze the security of construction Π_{POTS} when the seller $\mathcal{V}$ is corrupt.

C.1 Security Analysis of Π_{POTS} when $\mathcal{B}$ is corrupt

We first describe the simulator $\mathcal{S}$ for the case in which the buyer $\mathcal{B}$ is corrupt. $\mathcal{S}$ simulates the protocol by running the seller's side of protocol Π_{POTS} and copies of the ideal functionalities involved.

- Upon receiving a message from $\mathcal{A}$, $\mathcal{S}$ uses the first field of that message to associate the message with one of the ideal functionalities $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$, or $\mathcal{F}_{\mathrm{NHCD}}$, and runs a copy of the corresponding functionality on input that message.
- When a copy of any of the functionalities $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$, or $\mathcal{F}_{\mathrm{NHCD}}$ sends a message to $\mathcal{B}$ or to $\mathcal{S}$, $\mathcal{S}$ forwards the output of the functionality to $\mathcal{A}$.
- When a copy of any of the functionalities $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$, or $\mathcal{F}_{\mathrm{NHCD}}$ sends a message to $\mathcal{V}$, $\mathcal{S}$ runs protocol $\mathit{\Pi}_{\mathrm{POTS}}$ for the seller on input that message, except when $\mathcal{F}_{\mathrm{NIC}}$ sends a delayed output $(\mathsf{com.setup.end}, sid, OK)$ to $\mathcal{V}$.
- When $\mathcal{F}_{\mathrm{POTS}}$ sends the message $(\mathsf{pot.init.sim}, sid, ep, N)$ to $\mathcal{S}$, $\mathcal{S}$ sends the message $(\mathsf{pot.init.rep}, sid, ep)$ to $\mathcal{F}_{\mathrm{POTS}}$.
- When $\mathcal{F}_{\mathrm{POTS}}$ sends the message $(\mathsf{pot.setupprices.sim}, sid, \langle p\rangle_{n=1}^{N_{max}})$ to $\mathcal{S}$, $\mathcal{S}$ sends the message $(\mathsf{pot.setupprices.rep}, sid)$ to $\mathcal{F}_{\mathrm{POTS}}$.
- When $\mathcal{F}_{\mathrm{POTS}}$ sends the message $(\mathsf{pot.updateprice.sim}, sid, qid, n, p)$ to the simulator $\mathcal{S}$, the simulator $\mathcal{S}$ sends the message $(\mathsf{pot.updateprice.rep}, sid, qid)$ to functionality $\mathcal{F}_{\mathrm{POTS}}$.
- When $\mathcal{F}_{\mathrm{POTS}}$ outputs the message $(\mathsf{pot.init.end}, sid, ep, N)$, $\mathcal{S}$ runs the protocol $\mathit{\Pi}_{\mathrm{POTS}}$ for the seller on input $(\mathsf{pot.init.ini}, sid, ep, \langle m_n\rangle_{n=1}^{N})$, where $\langle m_n\rangle_{n=1}^{N}$ are selected at random.
- When $\mathcal{F}_{\mathrm{POTS}}$ outputs the message $(\mathsf{pot.setupprices.end}, sid, \langle p_n\rangle_{n=1}^{N_{max}})$, $\mathcal{S}$ runs protocol $\mathit{\Pi}_{\mathrm{POTS}}$ for the seller on input $(\mathsf{pot.setupprices.ini}, sid, \langle p_n\rangle_{n=1}^{N_{max}})$.

- When $\mathcal{F}_{\mathrm{POTS}}$ outputs the message $(\mathsf{pot.updateprice.end}, sid, n, p)$, $\mathcal{S}$ runs protocol Π_{POTS} for the seller on input $(\mathsf{pot.updateprice.ini}, sid, n, p)$.
- When protocol Π_{POTS} for the seller outputs the message $(\mathsf{pot.deposit.end}, sid, dep)$, $\mathcal{S}$ sends the message $(\mathsf{pot.deposit.ini}, sid, dep)$ to $\mathcal{F}_{\mathrm{POTS}}$. When $\mathcal{F}_{\mathrm{POTS}}$ sends the message $(\mathsf{pot.deposit.sim}, sid, qid)$ to $\mathcal{S}$, $\mathcal{S}$ sends $(\mathsf{pot.deposit.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{POTS}}$.
- When protocol Π_{POTS} outputs the message $(\mathsf{pot.transfer.end}, sid, m_\sigma)$, the simulator $\mathcal{S}$ retrieves the message $(\mathsf{ot.request.ini}, sid, \sigma, ccom_\sigma, copen_\sigma)$ sent by the adversary $\mathcal{A}$ to the ideal functionality $\mathcal{F}_{\mathrm{OT}}$. The simulator $\mathcal{S}$ sets $m'_\sigma \leftarrow m_\sigma$ for the copy of $\mathcal{F}_{\mathrm{OT}}$ associated with sid'_{OT}, such that $sid'_{\mathrm{OT}} \in (sid, ep)$. The simulator $\mathcal{S}$ sends the message $(\mathsf{pot.transfer.ini}, sid, ep, \sigma)$ to the functionality $\mathcal{F}_{\mathrm{POTS}}$. When $\mathcal{F}_{\mathrm{POTS}}$ sends the message $(\mathsf{pot.transfer.sim}, sid, qid, ep)$ to $\mathcal{S}$, $\mathcal{S}$ sends $(\mathsf{pot.transfer.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{POTS}}$.
- When protocol Π_{POTS} for the seller outputs $(\mathsf{pot.revealstatistic.end}, sid, v, \mathsf{ST})$, $\mathcal{S}$ sends the message $(\mathsf{pot.revealstatistic.ini}, sid, \mathsf{ST})$ to $\mathcal{F}_{\mathrm{POTS}}$. When $\mathcal{F}_{\mathrm{POTS}}$ sends the message $(\mathsf{pot.revealstatistic.sim}, sid, qid)$ to $\mathcal{S}$, $\mathcal{S}$ sends the message $(\mathsf{pot.revealstatistic.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{POTS}}$.
- $\mathcal{S}$ outputs failure if $\mathcal{A}$ produces two openings for a commitment.

Theorem 2. *When the buyer $\mathcal{B}$ is corrupt, the construction Π_{POTS} described in Section C.1 securely realizes $\mathcal{F}_{\mathrm{POTS}}$ in the $(\mathcal{F}_{\mathrm{AUT}}, \mathcal{F}_{\mathrm{SMT}}, \mathcal{F}_{\mathrm{NIC}}||\mathcal{S}_{\mathrm{NIC}}, \mathcal{F}_{\mathrm{ZK}}^{R}, \mathcal{F}_{\mathrm{OT}}, \mathcal{F}_{\mathrm{CD}}, \mathcal{F}_{\mathrm{NHCD}})$-hybrid model.*

Proof of Theorem 2. We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\mathrm{REAL}_{\Pi_{\mathrm{POTS}},\mathcal{A},\mathcal{Z}}$ and the ensemble $\mathrm{IDEAL}_{\mathcal{F}_{\mathrm{POTS}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote by Pr [**Game** i] the probability that the environment distinguishes **Game** i from the real-world protocol.

Game 0**:** This game corresponds to the execution of the real-world protocol. Therefore, Pr [**Game** 0] $= 0$.

Game 1**:** This game proceeds as **Game** 0, except that **Game** 1 replaces the messages $\langle m \rangle_{n=1}^{N_{max}}$ that are sent as input to the $\mathsf{ot.init}$ interface by random messages. This change does not alter the view of the environment because, in the $\mathsf{ot.init}$ interface, $\mathcal{F}_{\mathrm{OT}}$ does not send the messages to the simulator or to the buyer. Moreover, **Game** 1 copies the correct message m_σ to the copy of $\mathcal{F}_{\mathrm{OT}}$ before the $\mathsf{ot.transfer}$ interface is executed, so the corrupt buyer receives the correct message. Hence, [Pr [**Game** 1] $-$ Pr [**Game** 0]] $= 0$.

Game 2**:** This game proceeds as **Game** 1, except for the fact that **Game** 2 outputs failure when the adversary produces two openings for the same commitment. However, the probability that the adversary may produce two such openings is negligible, thanks to the binding property enforced by $\mathcal{F}_{\mathrm{NIC}}$. Therefore, [Pr [**Game** 2] $-$ Pr [**Game** 1]] $= 0$.

$\mathcal{S}$ is indistinguishable from the real world protocol because it runs as the real-world protocol, except when it outputs failure. The probability that $\mathcal{S}$ outputs failure is negligible thanks to the security properties ensured by the functionalities $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$, and $\mathcal{F}_{\mathrm{NHCD}}$. $\mathcal{F}_{\mathrm{NIC}}$ serves to ensure that all commitments used in the protocol are binding, while $\mathcal{F}_{\mathrm{ZK}}^{R}$ guarantees that the witness and instance values provided by the buyer for the deposit, transfer, and revealstatistic phases of the protocol satisfy the relations R_{dep}, R_{trans} and R_{count}, and R_{ST} respectively. R_{dep} is used to check whether the buyer is updating deposit values correctly during the deposit phase, while also ensuring that the commitments $ccom_{dep}$, $ccom_{dep_1}$, and $ccom_{dep_2}$ commit to the deposit value dep, the initial deposit dep_1, and the final deposit value dep_2. The protocol relies on R_{trans} to ensure that the buyer updates deposit and counter values consistently after a purchase, while also ensuring that the commitments to the message price p_σ, initial and final deposit values dep_1 and dep_2, counter index σ, and initial and final counter values $count_1$ and $count_2$ are valid. R_{ST} is used to make sure that the result of the evaluation of function ST on Tbl_{cd} is accurate, and that the commitments to all the indices and values of the counters in Tbl_{cd} used to determine the result of $\mathsf{ST}(\mathsf{Tbl}_{cd})$ are valid. $\mathcal{F}_{\mathrm{CD}}$ ensures that the buyer's deposit value dep and all counter values $\langle count_n \rangle_{n=1}^{N_{max}}$ read from positions in $\mathsf{Tbl}_{\mathcal{S}}$ are equal to the values previously written to those positions. $\mathcal{F}_{\mathrm{NHCD}}$ is used to ensure that the buyer can prove that she is using the right prices for the right messages (p_σ is used for m_σ). Finally, $\mathcal{F}_{\mathrm{OT}}$ ensures sender security (the buyer does not learn any information about messages that have not been purchased).

The distribution of **Game** 2 is identical to that of our simulation. This concludes the proof of theorem 2.

C.2 Security Analysis of $\Pi_{\mathbf{POTS}}$ when $\mathcal{V}$ is corrupt

In this section, we describe the simulator $\mathcal{S}$ for the case in which the seller $\mathcal{V}$ is corrupt. $\mathcal{S}$ simulates the protocol by running copies of the ideal functionalities and protocol Π_{POTS} for the buyer.

- Upon receiving a message from $\mathcal{A}$, $\mathcal{S}$ uses the first field of that message to associate the message with one of the ideal functionalities $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$, or $\mathcal{F}_{\mathrm{NHCD}}$, and runs a copy of the corresponding functionality on input that message.
- When a copy of any of the functionalities $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$, or $\mathcal{F}_{\mathrm{NHCD}}$ sends a message to $\mathcal{V}$ or to $\mathcal{S}$, $\mathcal{S}$ forwards the output of the functionality to $\mathcal{A}$, except when $\mathcal{F}_{\mathrm{NIC}}$ outputs a delayed message $(\mathsf{com.setup.end}, sid, OK)$.
- When a copy of any of the functionalities $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$, or $\mathcal{F}_{\mathrm{NHCD}}$ sends a message to $\mathcal{B}$, $\mathcal{S}$ runs protocol $\mathit{\Pi}_{\mathrm{POTS}}$ for the buyer on input that message.
- When protocol $\mathit{\Pi}_{\mathrm{POTS}}$ for the buyer sends a message to any of the functionalities $\mathcal{F}_{\mathrm{AUT}}$, $\mathcal{F}_{\mathrm{SMT}}$, $\mathcal{F}_{\mathrm{NIC}}$, $\mathcal{F}_{\mathrm{ZK}}^{R}$, $\mathcal{F}_{\mathrm{OT}}$, $\mathcal{F}_{\mathrm{CD}}$, or $\mathcal{F}_{\mathrm{NHCD}}$, $\mathcal{S}$ runs the copy of the respective functionality on input that message.
- When $\mathcal{F}_{\mathrm{POTS}}$ sends the message $(\mathsf{pot.deposit.sim}, sid, qid)$ to $\mathcal{S}$, $\mathcal{S}$ sends the message $(\mathsf{pot.deposit.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{POTS}}$.
- When $\mathcal{F}_{\mathrm{POTS}}$ sends the message $(\mathsf{pot.revealstatistic.sim}, sid, qid)$ to $\mathcal{S}$, $\mathcal{S}$ sends the message $(\mathsf{pot.revealstatistic.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{POTS}}$.
- When $\mathcal{F}_{\mathrm{NIC}}$ outputs the message $(\mathsf{com.setup.req}, sid, qid)$, $\mathcal{S}$ runs a copy of $\mathcal{S}_{\mathrm{NIC}}$ on input that message. When $\mathcal{S}_{\mathrm{NIC}}$ replies with $(\mathsf{com.setup.alg}, sid, qid, m)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NIC}}$ on input that message.

- When protocol Π_{POTS} for the buyer outputs the message $(\mathsf{pot.init.end}, sid, ep, N)$, $\mathcal{S}$ sets $sid'_{\text{OT}} \leftarrow (sid, ep)$, and retrieves $(\mathsf{ot.init.ini}, sid_{\text{OT}}, \langle m_n \rangle_{n=1}^{N})$ sent by $\mathcal{A}$ to $\mathcal{F}_{\text{OT}}$ such that $sid'_{\text{OT}} = sid_{\text{OT}}$, and sends the message $(\mathsf{pot.init.ini}, sid, ep, \langle m_n \rangle_{n=1}^{N})$ to $\mathcal{F}_{\text{POTS}}$. When $\mathcal{F}_{\text{POTS}}$ sends the message $(\mathsf{pot.init.sim}, sid, ep, N)$ to $\mathcal{S}$, $\mathcal{S}$ sends the message $(\mathsf{pot.init.rep}, sid, ep)$ to $\mathcal{F}_{\text{POTS}}$.
- When protocol Π_{POTS} for the buyer outputs the message $(\mathsf{pot.setupprices.end}, sid, \langle p_n \rangle_{n=1}^{N_{max}})$, $\mathcal{S}$ sends the message $(\mathsf{pot.setupprices.ini}, sid, \langle p_n \rangle_{n=1}^{N_{max}})$ to the functionality $\mathcal{F}_{\text{POTS}}$. When $\mathcal{F}_{\text{POTS}}$ sends the message $(\mathsf{pot.setupprices.sim}, sid, \langle p_n \rangle_{n=1}^{N_{max}})$ to $\mathcal{S}$, $\mathcal{S}$ sends the message $(\mathsf{pot.setupprices.rep}, sid)$ to $\mathcal{F}_{\text{POTS}}$.
- When protocol Π_{POTS} for the buyer outputs the message $(\mathsf{pot.updateprice.end}, sid, n, p)$, $\mathcal{S}$ sends the message $(\mathsf{pot.updateprice.ini}, sid, n, p)$ to $\mathcal{F}_{\text{POTS}}$.
- When $\mathcal{F}_{\text{POTS}}$ sends $(\mathsf{pot.updateprice.sim}, sid, qid, n, p)$ to $\mathcal{S}$, $\mathcal{S}$ sends the message $(\mathsf{pot.updateprice.rep}, sid, qid)$ to $\mathcal{F}_{\text{POTS}}$.
- When $\mathcal{F}_{\text{POTS}}$ outputs the message $(\mathsf{pot.deposit.end}, sid, dep)$, $\mathcal{S}$ sends the message $(\mathsf{pot.deposit.ini}, sid, dep)$ to protocol Π_{POTS} for the buyer.
- $\mathcal{S}$ selects σ such that σ is associated with the message with the lowest price p_σ and sends the message $(\mathsf{pot.transfer.ini}, sid, ep, \sigma)$ to Π_{POTS} for the buyer.
- When $\mathcal{F}_{\text{POTS}}$ outputs the message $(\mathsf{pot.revealstatistic.end}, sid, v, \mathsf{ST})$, $\mathcal{S}$ sends the message $(\mathsf{pot.revealstatistic.ini}, sid, \mathsf{ST})$ to protocol Π_{POTS} for the buyer.

Theorem 3. *When the seller $\mathcal{V}$ is corrupt, the construction Π_{POTS} described in Section C.1 securely realizes $\mathcal{F}_{\text{POTS}}$ in the $(\mathcal{F}_{\text{AUT}}, \mathcal{F}_{\text{SMT}}, \mathcal{F}_{\text{NIC}}||\mathcal{S}_{\text{NIC}}, \mathcal{F}_{\text{ZK}}^{R}, \mathcal{F}_{\text{OT}}, \mathcal{F}_{\text{CD}}, \mathcal{F}_{\text{NHCD}})$-hybrid model.*

Proof of Theorem 3. We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\text{REAL}_{\Pi_{\text{POTS}},\mathcal{A},\mathcal{Z}}$ and the ensemble $\text{IDEAL}_{\mathcal{F}_{\text{POTS}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote by Pr [**Game** i] the probability that the environment distinguishes **Game** i from the real-world protocol.

Game 0**:** This game corresponds to the execution of the real-world protocol. Therefore, Pr [**Game** $0] = 0$.

Game 1**:** This game proceeds as **Game** 0, except that in **Game** 1, the simulator simulates the buyer's side of the protocol by selecting σ associated with the message with the lowest price. This does not alter the view of the environment. Therefore, $[\Pr[$**Game** $1] - \Pr[$**Game** $0] = 0]$.

Our simulator $\mathcal{S}$ is indistinguishable from the real world protocol because it runs as the real-world protocol, except when it outputs failure. The probability that $\mathcal{S}$ outputs failure is negligible thanks to the security properties ensured by the functionalities $\mathcal{F}_{\text{AUT}}$, $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{NIC}}$, $\mathcal{F}_{\text{ZK}}^{R}$, $\mathcal{F}_{\text{OT}}$, $\mathcal{F}_{\text{CD}}$, and $\mathcal{F}_{\text{NHCD}}$. Concretely, $\mathcal{F}_{\text{NIC}}$ ensures that commitments are binding, and $\mathcal{A}$ does not learn any information on σ, thanks to $\mathcal{F}_{\text{OT}}$. The obliviousness property provided by $\mathcal{F}_{\text{CD}}$ also ensures that $\mathcal{A}$ does not learn any information on the counter values $count_n$ and deposit value dep stored in Tbl_{cd}.

The distribution of **Game** 1 is identical to that of our simulation. This concludes the proof of theorem 3.