Multiparty Computation Protocols for Private Compliance Checks

Silence Laboratories
info@silencelaboratories.com

November 2, 2024

Abstract

This document provides the technical specifications of the Multiparty Computation (MPC) protocols used to perform compliance checks on bank transaction. There are two types of checks implemented: Private Sanctions List compliance, and Capital Flow Management. The former check ensures that a transaction does not involve entities present on a private sanction list, and the latter verifies that executing the transaction will not induce a customer's total capital flows to exceed a specified limit. The security model and MPC protocol descriptions relevant to these tasks are given in this document.

Contents

1 Introduction **1**
- 1.1 Functionality Overview 1
- 1.2 Tools Used 3

2 Preliminaries **4**

3 Sanctions List Check **4**

4 Capital Flow Management **7**
- 4.1 Building Blocks 9
- 4.2 Main Protocol 18

1 Introduction

This document specifies the protocols to be invoked for the Private Sanctions List check, as well as the Capital Flow Management (CFM) measures check. As the two are distinct tasks with their own challenges, we separate the corresponding protocols and their analyses. Each protocol provides security against a malicious adversary that may arbitrarily deviate from the protocol, in the standard real-ideal paradigm for MPC protocols [Can00, Can01]. This paradigm implies that each protocol emulates an ideal oracle which takes private inputs from each party, performs a specified computation on their union, and returns only the result with no leakage of intermediate state. Note that this oracle does not at the moment bind the inputs to any external data sources, as input validation is considered out of scope for this project. We specify the structure of inputs, outputs, and the exact computation in the relevant sections, and give only a brief overview below.

1.1 Functionality Overview

The Private Sanctions List check allows for an originating bank OB of a transaction to verify that the beneficiary does not appear in a list of sanctioned entities held by the Beneficiary Bank BB. The protocol protects each party's private inputs: BB learns absolutely no information about the transaction, and OB learns only of the presence (or absence) of the beneficiary on the sanctions list. For instance, in the event that the beneficiary is present on the sanctions list, the names of other entities on the list appear as unintelligible pseudorandom values to BB. We depict the information flow of this functionality in Figure 1.

The Capital Flow Management protocol enables an originating bank OB of a transaction to verify that its execution will not induce the beneficiary's balance to exceed a specified limit. The Central Bank CB maintains a list of customer balances in a private database, and the protocol ensures that *only* the predicate relevant to this computation is revealed. In particular, CB learns no information about the transaction—the amount or the identities of parties involved—and OB does not learn any new information about the beneficiary's existing capital flows beyond the outcome of the check. In particular, all information about CB's private database (including the identities and capital flows of other customers) is completely hidden from OB. We depict the information flow of this functionality in Figure 2.

Note that the validity of the check is only guaranteed if there are no changes to the beneficiary's balance between the time the protocol is instantiated and the time that the transaction is executed. Therefore, it is recommended that the protocol query and its corresponding transaction (if valid) be executed atomically when possible.

Both protocols have mechanisms in place to detect, and protect against active deviations. The only attack that a corrupt party can execute is that of denial of service by withholding its messages, or inducing an abort by sending a malformed one. It is possible in the Capital Flow Management protocol that

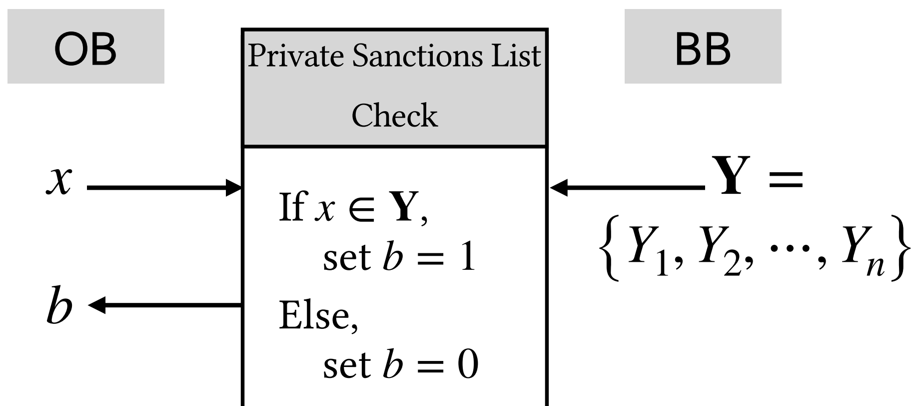

Figure 1: Functionality provided by the Private Sanctions List check, in which the Originating Bank OB provides the beneficiary name x while Beneficiary Bank BB provides its private list of sanctioned entities $\mathbf{Y}$. The MPC securely returns a bit that indicates the presence/absence of x in the list $\mathbf{Y}$.

the beneficiary queried by OB is not one of the customers in CB's database, in which case the protocol also aborts. The formal definitions of each functionality can be found in the relevant sections.

Caveats of non-validation of inputs. As inputs are not bound to any external sources in this system, it is important to consider that an invalid input by one party may inadvertently induce some leakage of the other party's private inputs—this is by virtue of the ideal functionality itself rather than protocol shortcomings. For instance in the sanctions list check, BB may choose to omit a subset of its true sanctions list to test if OB's input lies outside of that subset. In the capital flow management check, CB may provide an invalid balance as the entry for a specific customer, to test if the MPC attempts to read that customer's balance due to OB's input—commonly called a "selective failure" attack. Another consequence of not validating inputs is that desired outputs can be induced by deliberately chosen inputs—for instance BB may supply as its input to the sanctions list check a meaningless string that is highly unlikely to correspond to any real entity (sanctioned or otherwise)—this makes certified outputs only as trustworthy as the participants executing the check. Once more, these issues stem from the fundamental nature of the task at hand rather than any aspects specific to the protocols in this document.

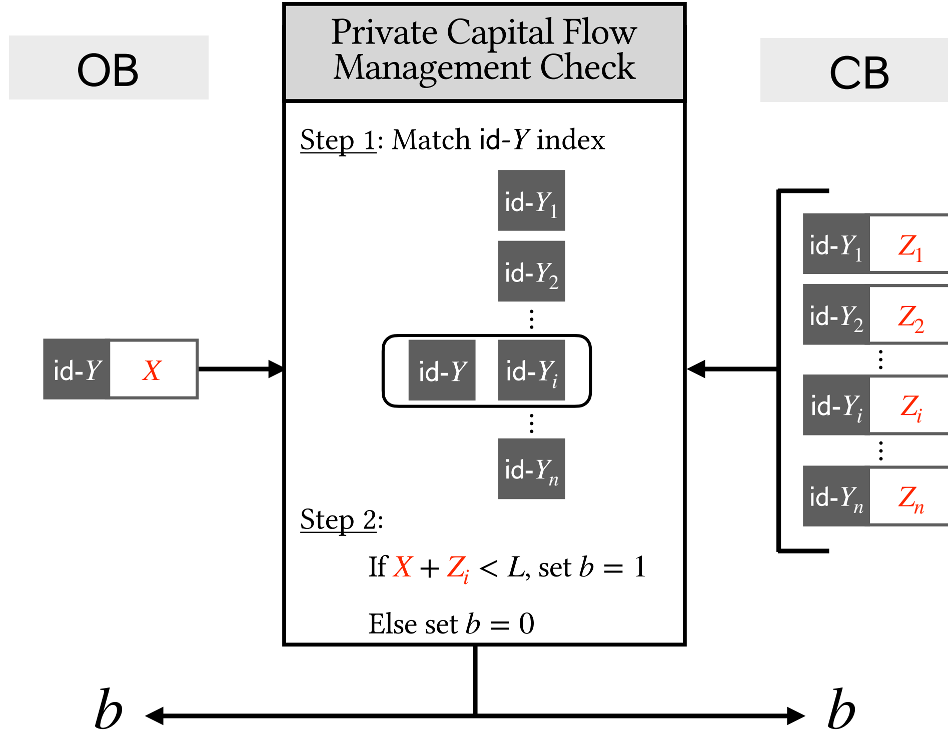

Figure 2: Functionality provided by the Private Capital Flow Management check, in which the Originating Bank OB provides the company details Y and proposed transaction amount X, while Central Bank CB provides its table of sum of existing capital flows for all customers. The MPC securely reads the entry Z_i from CB's table that corresponds to OB's input Y, checks if the sum $X + Z_i$ exceeds the specified limit L, and finally reports only the outcome of this comparison to both parties.

1.2 Tools Used

The secure computation task that underlies the Private Sanctions List check is a Private Set Intersection (PSI), which is a well-studied problem in the MPC literature. Our protocol employs a classic Diffie-Hellman based PSI technique, which can be realized with only an elliptic curve and a hash function.

The Capital Flow Management check is more complex, requiring general purpose MPC tools. In particular, securely comparing private values is at the heart of the protocol, and known techniques for this task involve effectively computing the comparison circuit using MPC. General MPC, particularly with security against active protocol deviations, requires composing several advanced crypto-

graphic tools. We follow a standard recipe for designing such protocols laid out in works such as BDOZ [BDOZ11], SPDZ [DPSZ12], and MASCOT [KOS16]: an Oblivious Linear Evaluation (OLE) is used to generate Beaver Triples along with corresponding Message Authentication Codes (MACs), which are then used to securely evaluate an arithmetic circuit in topological order. We make use of the arithmetic recurrence relation formulated by Garay et al. [GSV07] to effectively implement a comparison circuit. Informed by efficiency tradeoffs established in the context of a widely deployed MPC-based tool—ECDSA signing—we utilize Oblivious Linear Evaluation (OLE) based on Oblivious Transfer (and its Extension [IKNP03, KOS15, Roy22]), due to its relatively low computational footprint. Our resulting protocol can be thought of as MASCOT [KOS16] with slightly reduced efficiency, but adjusted abstractions to incorporate audited libraries.

This document only specify protocols as necessary for implementation, with little expository text. As such, the reader is assumed to have a high degree of expertise in Multiparty Computation protocols, and is encouraged to contact the authors to clarify points of ambiguity.

We begin with the Sanctions List check protocol in Section 3. The Capital Flow Management protocol is given in Section 4.

2 Preliminaries

We adopt the real-ideal paradigm as is standard for MPC protocols. When the underlying primitives are UC-secure [Can01], our protocols achieve UC security as well. Notation and mathematical details will be introduced as relevant in each protocol or functionality. All functionalities in this paper are run by two parties, and therefore all outputs are adversarially delayed as is inherent in this setting [Cle86]. We use standard elliptic curve notation, wherein a curve $\mathbb{G}$ of prime order q is generated by $G \in \mathbb{G}$,

3 Sanctions List Check

We first formalize the problem by means of an ideal functionality $\mathcal{F}_{\mathsf{PSC}}$. The functionality is invoked by two parties, the *beneficiary bank*, denoted BB, and the *originating bank*, denoted OB. The functionality $\mathcal{F}_{\mathsf{PSC}}$ allows BB and OB to privately supply as inputs a list of customers $\mathbf{Y}$, and an individual customer x respectively. Upon receiving these inputs, $\mathcal{F}_{\mathsf{PSC}}$ informs OB of the presence of x in $\mathbf{Y}$. The only information leaked to OB besides the predicate $x \in \mathbf{Y}$ is the size of the list $|\mathbf{Y}|$, while BB gets no information about x whatsoever. We give the functionality below.

Functionality 3.1. $\mathcal{F}_{\mathsf{PSC}}(1^\lambda)$: Private Sanction List Check

This functionality is accessed by parties BB and OB, and the sanctions list size is a fixed parameter. Upon receiving $(\mathsf{sid}, \texttt{sanction-list}, \mathbf{Y})$ from

BB and $(\mathsf{sid}, \mathtt{customer}, x)$ from OB such that sid is fresh and $\mathbf{Y}$ is a list of values of the correct size, return (sid, b) to OB, where b is a bit such that $b = 1$ iff $x \in \mathbf{Y}$.

A subtle technical point is that our functionality permits *delayed extraction* of OB's input, as BB is not notified when OB supplies its input. This is due to a common issue with modelling standard Oblivious PRF constructions. As in the OPRF protocols, OB is de-facto committed to its input once it sends its protocol message, just that we are unable to extract it until it wishes to claim its output. We now give our two-party protocol π_{PSC} to realize the above functionality. Our protocol is derived from classic private set intersection protocols based on the Diffie-Hellman problem [BKM+20, HFH99]. It requires two messages:

1. BB samples a key k for a pseudorandom function, and evaluates this function on each element of $\mathbf{Y}$ to derive $\hat{\mathbf{Y}}$—a pseudorandom encoding of the sanctions list.
2. OB and BB engage in an Oblivious PRF protocol, so that OB securely obtains $\hat{x}$—the evaluation of x as per BB's PRF key.
3. BB sends $\hat{\mathbf{Y}}$ to OB, who determines if $x \in \mathbf{Y}$ iff $\hat{x} \in \hat{\mathbf{Y}}$.

We instantiate the PRF with standard Diffie-Hellman based techniques using elliptic curves, in a way that supports efficient oblivious evaluation. The full protocol requires only one simple zero-knowledge proof to guarantee malicious security, which we model via the $\mathcal{F}_{\mathsf{DL}}$ functionality below.

Functionality 3.2. $\mathcal{F}_{\mathsf{DL}}(\mathbb{G}, q)$**: PoK of Discrete Logarithm**

This functionality is accessed by a prover and verifier. Upon receiving $(\mathsf{sid}, \mathtt{prove}, G, X, x)$ from the prover, where sid is fresh, and $G, X \in \mathbb{G}$, send $(\mathsf{sid}, \mathtt{proven}, G, X, b)$ to the verifier where the bit $b = 1$ iff $x \cdot G \stackrel{?}{=} X$.

With the basic structure and all building blocks in place, we give the full protocol itself below.

Protocol 3.3. $\pi_{\mathsf{PSC}}(1^\lambda)$**: Private Sanction List Check**

This protocol is parameterized by the originating bank OB, the beneficiary bank BB, and the elliptic curve $\mathcal{G} = (\mathbb{G}, G, q)$. The protocol runs between the parties OB and BB, of which any one may be *maliciously* corrupt. The private input of this protocol for BB is a *sanctions list* $\mathbf{Y}$ and for OB is the details of the *customer* x. The private output of the protocol for OB is a boolean value b, where b is *true* if customer x is present on list $\mathbf{Y}$, *false* otherwise.

This protocol functions in the $\mathcal{F}_{\mathsf{DL}}$-hybrid model, and makes use of hash functions $H_1 : \{0,1\}^* \to \mathbb{G}$ and $H_2 : \{0,1\}^* \to \{0,1\}^{2\lambda}$, modelled as random oracles. Note that H_1 hashes directly to a curve point $\mathbb{G}$, keeping the discrete logarithm of the point hidden.

Sanction Check.:

1. On input $(\mathsf{sid}, \mathtt{customer}, x)$, party OB does the following:
   (a) Sample a random element $r \in \mathbb{Z}_q$, and compute $A = r \cdot H_1(x)$.
   (b) Send (sid, A) to BB.
2. On receiving (sid, A) from OB, and private input $(\mathsf{sid}, \mathtt{sanction\text{-}list}, \mathbf{Y})$, party BB does the following:
   (a) Sample $k \leftarrow \mathbb{Z}_q$ and compute $B = k \cdot A$.
   (b) Sample a random permutation $\Pi : |\mathbf{Y}| \mapsto |\mathbf{Y}|$, and generate encoded list $\hat{\mathbf{Y}}$ as follows:
   $$\hat{\mathbf{Y}} = \{H_2(H_1(Y), k \cdot H_1(Y))\}_{Y \in \Pi(\mathbf{Y})}$$
   (c) Send $(\mathsf{sid}, \hat{\mathbf{Y}}, B)$ to OB, and $(\mathsf{sid}, \mathtt{prove}, A, B, k)$ to $\mathcal{F}_{\mathsf{DL}}$.
3. On receiving $(\mathsf{sid}, \hat{\mathbf{Y}}, B)$ from BB and $(\mathsf{sid}, A, B, 1)$ from $\mathcal{F}_{\mathsf{DL}}$, OB does the following:
   (a) Compute $\hat{x} = H_2(H_1(x), r^{-1} \cdot \mathsf{B})$
   (b) Check if $\hat{x} \in \hat{\mathbf{Y}}$. If so, set $b = 1$, otherwise set $b = 0$.
   (c) Output b.

We prove security of our protocol with a proof strategy that borrows heavily from Jarecki et al. [JKKX16], including a special case of their hardness assumption (one-more Diffie-Hellman, as commonly used in OPRF protocols).

Theorem 3.4. *Assuming the hardness of the* $(2, 1)$ *one-more Diffie-Hellman (OMDH) problem in* $\mathbb{G}$*, the protocol* π_{PSC} *realizes functionality* $\mathcal{F}_{\mathsf{PSC}}$ *in the local random oracle and* $\mathcal{F}_{\mathsf{DL}}$*-hybrid model, in the presence of up to one malicious static corruption.*

Proof. (Sketch) We describe here how to simulate the protocol, first in the presence of a corrupt BB, and then in the presence of a corrupt OB, and briefly argue indistinguishability of the simulation.

Corrupt BB. The only value sent by OB to BB is A. Given that r is sampled uniformly from $\mathbb{Z}_q$ and does not occur anywhere else in BB's view, A is trivial to simulate by sending a uniformly random $\mathbb{G}$ element. Next, we show how the simulator can extract BB's input $\mathbf{Y}$ to send to $\mathcal{F}_{\mathsf{PSC}}$. Observe by the structure of Step 3a that OB will only match with elements in $\hat{\mathbf{Y}}$ that are of the form $H_2(H_1(Y), k \cdot H_1(Y))$, where k is the same as the one sent to $\mathcal{F}_{\mathsf{DL}}$. Given that the ranges of H_1,H_2 are exponentially large, with overwhelming probability each Y (resp. $H_1(Y), k \cdot H_1(Y)$) must be in the list of queries to H_1 (resp. H_2), and the simulator can simply search for them, and assemble $\mathbf{Y}$ to send to $\mathcal{F}_{\mathsf{PSC}}$.

Corrupt OB**.** The simulator begins by executing Step 2a of π_{PSC} to generate k, B. However, it samples $\hat{\mathbf{Y}} \leftarrow \{0,1\}^{2\lambda \times |\mathbf{Y}|}$ uniformly at random, and sends $B, \hat{\mathbf{Y}}$ to BB. Note at this point that this simulation is distributed identically to the real protocol, if no $H_1(x), k \cdot H_1(x)$ is queried to H_2. We argue that there is at most one such query that is made by the corrupt OB. First note that upon this query being made, the corresponding x can be forwarded by the simulator to $\mathcal{F}_{\mathsf{PSC}}$ as OB's input, and if $\mathcal{F}_{\mathsf{PSC}}$ responds with $(\mathsf{sid}, 1)$, then the simulator can program H_2 so that $H_2(H_1(x), k \cdot H_1(x))$ matches a randomly chosen element of $\hat{\mathbf{Y}}$. If the corrupt OB is able to make a single extra well-formed query that *matches* any element Y of the honest BB's input $\mathbf{Y}$, then we can construct a reduction to $(2,1)$ OMDH that succeeds with a polynomially related probability.

Recall that in the $(2,1)$ OMDH assumption, the challenger provides $G, k \cdot G, G_1, G_2$ and the opportunity to make one query to an oracle $\mathcal{O}(k, \cdot)$ that returns the exponentiation (i.e. curve multiplication) of the query by k; the adversary's task is to return $k \cdot G_1$ and $k \cdot G_2$. The reduction works as follows: upon receiving the OMDH challenge, choose two random elements $Y_1, Y_2 \in \mathbf{Y}$ (this is available to the reduction as the distinguisher must specify the honest party's input) and program $H_1(Y_1) = G_1$ and set $H_1(Y_2) = G_2$. Upon receiving A from the corrupt OB, query $\mathcal{O}(k, A)$ and use its response as B in the message to OB. Subsequently, search OB's list of queries to H_2 and choose any two queries at random that are of the form $(H_1(Y_1), \alpha)$ and $(H_1(Y_2), \beta)$, programming the earlier of the two to match a random element of $\hat{\mathbf{Y}}$. Send α, β to the challenger, and halt. Observe that this reduction solves the challenge successfully when its choice of Y_1, Y_2 corresponds to the two well-formed queries made by the corrupt OB, and its choice of queries made to H_2 are in fact the ones that are correct (i.e. well-formed). The first choice is successful with probability $1/|\mathbf{Y}|^2$, and the second is successful with probability $1/Q^2$, where Q is the number of queries that the corrupt OB makes to the random oracle (which is polynomially bounded). The overall probability of success is therefore only a polynomial factor less than the success of a corrupt OB making two well-formed queries that match $\mathbf{Y}$ values, implying that this event must occur with only negligible probability assuming that the $(2,1)$ OMDH problem is hard. □

4 Capital Flow Management

The Capital Flow Management protocol is used by an Originating Bank OB that holds a proposed transaction amount X for a company Y, and a Central Bank CB that holds a list of existing capital flows $\mathbf{Z}$ for a range of companies. We assume that each entry of $\mathbf{Z}$ records the latest balance of each corresponding company, i.e. all transactions relevant to a company have already been locally aggregated on CB's end. Note that we wish to keep $\mathbf{Z}$ fully encrypted through the protocol, including identifying information about companies not involved in this transaction.

The protocol allows OB to check if X for a company Y will, in combination with its existing capital flows $\mathbf{Z}_Y$, exceed a specified limit L. Importantly, each

party's private input must stay hidden from the other. This task involves two non-trivial secure computation components: the first is for OB to retrieve $\mathbf{Z}_Y$ in encrypted form without revealing Y or the names of other companies for which CB maintains data, and the second is to check if $\mathbf{Z}_Y$ in combination with X exceeds L, while keeping $\mathbf{Z}_Y$ and X encrypted.

We make use of the following building block.

Functionality 4.1. $\mathcal{F}_{\mathsf{PSt}}(1^\lambda)$**: Private Set Intersection and Transfer**

This functionality is accessed by parties CB and OB, and the capital flows list size is a fixed parameter. Upon receiving $(\mathsf{sid}, \texttt{masked-capital-flows-list}, \mathbf{Y}, \{\mathbf{Z}_y^{\mathsf{OB}}\}_{y\in\mathbf{Y}}, \{\mathbf{M}_y\}_{y\in\mathbf{Y}})$ from CB and $(\mathsf{sid}, \texttt{company}, Y)$ from OB such that sid is fresh and $\mathbf{Y}$ is a list of values of the correct size, return $(\mathsf{sid}, \mathbf{Z}_Y^{\mathsf{OB}}, \mathbf{M}_Y)$ to OB, where $\mathbf{Z}_Y^{\mathsf{OB}} = \perp$, $\mathbf{M}_Y = \perp$ if $Y \notin \mathbf{Y}$.

Our approach can be outlined as follows:

- Let B be the bit length of the integer datatype in this context, and let λ_{s} be a statistical security parameter. One can think of B as say 47 bits, and λ_{s} as 80 bits. Define $\ell = B + \lambda_{\mathsf{s}} + 1$.
- CB holds a list of company names $\mathbf{Y}$, and each $y \in \mathbf{Y}$ is accompanied by an integer $\mathbf{Z}_y$ that represents its balance. CB samples a random mask $Z^{\mathsf{CB}} \leftarrow \mathbb{Z}_{2^{\ell-1}}$, and encrypts each entry of $\mathbf{Z}$ with this mask, to derive $\mathbf{Z}^{\mathsf{OB}} = \{\mathbf{Z}_y + Z^{\mathsf{CB}}\}_{y\in\mathbf{Y}}$.
- CB and OB invoke $\mathcal{F}_{\mathsf{PSt}}$ with private inputs $\mathbf{Y}, \mathbf{Z}^{\mathsf{OB}}$ and Y respectively, so that OB obtains $\mathbf{Z}_Y^{\mathsf{OB}}$.
- OB and CB engage in a secure comparison protocol with private inputs $X + \mathbf{Z}_Y^{\mathsf{OB}}$ and $L + Z^{\mathsf{CB}}$ respectively. Note that X pushes Y's balance over the limit iff $X + \mathbf{Z}_Y^{\mathsf{OB}} > L + Z^{\mathsf{CB}}$.

We now construct a two-party protocol π_{PSt} to realize the above functionality.

Protocol 4.2. $\pi_{\mathsf{PSt}}(1^\lambda)$**: Private Set Intersection and Transfer**

This protocol is parameterized by the central bank CB, the originating bank OB, and the elliptic curve $\mathcal{G} = (\mathbb{G}, G, q)$. The protocol runs between the parties CB and OB, of which any one may be *maliciously* corrupt. The private input of this protocol for CB is a *list of companies* $\mathbf{Y}$, a *masked capital flows list* $\{\mathbf{Z}_y^{\mathsf{OB}}\}_{y\in\mathbf{Y}}$, a MAC list $\{\mathbf{M}_y\}_{y\in\mathbf{Y}}$ and for OB is the *company* Y. The private output of the protocol for OB is a masked value $\mathbf{Z}_Y^{\mathsf{OB}}$ with the corresponding MAC $\mathbf{M}_Y$, where $\mathbf{Z}_Y^{\mathsf{OB}} = \perp$, $\mathbf{M}_Y = \perp$ if $Y \notin \mathbf{Y}$.

This protocol functions in the $\mathcal{F}_{\mathsf{DL}}$-hybrid model, and makes use of hash functions $H_1 : \{0,1\}^* \to \mathbb{G}$, $H_2 : \{0,1\}^* \to \{0,1\}^{2\lambda}$ and $H_3 : \{0,1\}^* \to \{0,1\}^{2l}$, modeled as random oracles. Note that H_1 hashes directly to a

curve point $\mathbb{G}$, keeping the discrete logarithm of the point hidden.

Private Transfer.:

1. On input $(\mathsf{sid}, \mathtt{company}, Y)$, party OB does the following:
   (a) Sample a random element $r \in \mathbb{Z}_q$, and compute $A = r \cdot H_1(Y)$.
   (b) Send (sid, A) to CB.
2. On receiving (sid, A) from OB, and private input $(\mathsf{sid}, \mathtt{masked\text{-}capital\text{-}flows\text{-}list}, \mathbf{Y}, \{\mathbf{Z}_y^{\mathsf{OB}}\}_{y \in \mathbf{Y}}, \{\mathbf{M}_y\}_{y \in \mathbf{Y}})$, party CB does the following:
   (a) Sample $k \leftarrow \mathbb{Z}_q$ and compute $B = k \cdot A$.
   (b) Sample a random permutation $\Pi : |\mathbf{Y}| \mapsto |\mathbf{Y}|$, and generate encoded list $\hat{\mathbf{Y}}$, encrypted list $\hat{\mathbf{Z}}^{\mathsf{OB}}$ as follows:
   $$\hat{\mathbf{Y}} = \{H_2(H_1(y), k \cdot H_1(y))\}_{y \in \Pi(\mathbf{Y})}$$
   $$\hat{\mathbf{Z}}^{\mathsf{OB}} = \{H_3(H_1(y), k \cdot H_1(y)) \oplus (\mathbf{Z}_y^{\mathsf{OB}} || \mathbf{M}_y)\}_{y \in \Pi(\mathbf{Y})}$$
   (c) Send $(\mathsf{sid}, \hat{\mathbf{Y}}, \hat{\mathbf{Z}}^{\mathsf{OB}}, B)$ to OB, and $(\mathsf{sid}, \mathtt{prove}, A, B, k)$ to $\mathcal{F}_{\mathsf{DL}}$.
3. On receiving $(\mathsf{sid}, \hat{\mathbf{Y}}, \hat{\mathbf{Z}}^{\mathsf{OB}}, B)$ from BB and $(\mathsf{sid}, \mathtt{proven}, A, B, 1)$ from $\mathcal{F}_{\mathsf{DL}}$, OB does the following:
   (a) Compute $\hat{x} = H_2(H_1(x), r^{-1} \cdot \mathsf{B})$
   (b) Check if $\hat{x} \in \hat{\mathbf{Y}}$. If so, let $\hat{x}$ be present in the ith index of the list $\hat{\mathbf{Y}}$. Set $(\mathbf{Z}_Y^{\mathsf{OB}} || \mathbf{M}_Y) = \hat{\mathbf{Z}}_i^{\mathsf{OB}} \oplus H_3(H_1(x), r^{-1} \cdot \mathsf{B})$, otherwise set $\mathbf{Z}_Y^{\mathsf{OB}} = \perp$, $\mathbf{M}_Y = \perp$.
   (c) Output $(\mathbf{Z}_Y^{\mathsf{OB}}, \mathbf{M}_Y)$.

Theorem 4.3. *Assuming the hardness of the* $(2,1)$ *one-more Diffie-Hellman (OMDH) problem in* $\mathbb{G}$*, the protocol* π_{PSt} *realizes functionality* $\mathcal{F}_{\mathsf{PSt}}$ *in the local random oracle and* $\mathcal{F}_{\mathsf{DL}}$*-hybrid model, in the presence of up to one malicious static corruption.*

Proof. The protocol and functionality are structured essentially the same as in Theorem 3.4, and so we omit the proof. □

4.1 Building Blocks

We use the Arithmetic Black Box (ABB) abstraction [DN07] to significantly simplify the exposition of our protocols and proofs. We consider each secret-shared variable to be stored by a reactive functionality, $\mathcal{F}_{\mathsf{ABB}}$, also called the ABB. The ABB can perform basic operations on stored variables, including addition and

multiplication. It can also perform operations which can be represented using an arithmetic circuit, such as testing that a variable is a bit or evaluating a comparison on two bit-representations of integers. Finally, it allows parties to input private variables (corresponding to secret-sharing) and allows variables to be output to parties (corresponding to share reconstruction).

Functionality 4.4. $\mathcal{F}_{\mathsf{ABB}}$**: Arithmetic Black Box**

Init(p, η_I, η_M)**:** :

1. Initializes the Arithmetic Black Box to an empty namespace.
2. Assigns prime modulus p. All ABB operations are modulo p. For security, p should be a large (e.g. 128-bit) prime.
3. Specifies η_I, the number of variables which will be input, and η_M, the number of multiplications (between ABB-stored variables) which will be performed. For simplicity, these are not always stated explicitly, in which case they are implicitly determined by the number of inputs and multiplications later performed. (This allows our implementation to be efficient by generating correlated randomness for the Input and Multiply operations during an input-independent pre-processing stage.)

OB**-Input**(y)**:**

1. Variable, y, held privately by OB, is input into the ABB.
2. The new variable is assigned the label $[y]$.

CB**-Input**(y)**:**

1. Variable, y, held privately by CB, is input into the ABB.
2. The new variable is assigned the label $[y]$.

Open-to-CB **(**$[x]$**):**

1. Variable $[x]$, held in the ABB is revealed to CB.
2. The new private variable is assigned label x.

Open-to-OB **(**$[x]$**):**

1. Variable $[x]$, held in the ABB is revealed to OB.
2. The new private variable is assigned label x.

Open($[x]$**):**

1. Variable $[x]$, held in the ABB is revealed to both parties.

2. The new public variable is assigned label x.

Add-Const($[x], c$**):** Shorthand $[x] + c$.

1. Creates a new variable in the ABB which is the sum of $[x]$ (held by the ABB) and c (a public variable).

2. The variable is then assigned a label (if in an assignment clause) or is used in a computation (if in a compute clause).

Mult-Const($[x], c$**):** Shorthand $c \cdot [x]$.

1. Creates a new variable in the ABB which is the product of $[x]$ (held in the ABB) and c (a public variable).

2. The variable is then assigned a label (if in an assignment clause) or is used in a computation (if in a compute clause).

Add($[x], [y]$**):** Shorthand $[x] + [y]$.

1. Creates a new variable in the ABB which is the sum of $[x]$ and $[y]$ (both held in the ABB).

2. The variable is then assigned a label (if in an assignment clause) or is used in a computation (if in a compute clause).

Multiply($[x], [y]$**):** Shorthand $[x] \cdot [y]$.

1. Creates a new variable in the ABB which is the product of $[x]$ and $[y]$ (both held in the ABB).

2. The variable is then assigned a label (if in an assignment clause) or is used in a computation (if in a compute clause).

Random():

1. Create a new variable in the ABB randomly chosen from $\mathbb{Z}_p$.

2. The variable is then assigned a label (if in an assignment clause) or is used in a computation (if in a compute clause).

TestBit($[x]$**):**

1. Creates a new variable in the ABB holding the value $1-([x]\cdot(1-[x]))$. Note that if $x \in \{0, 1\}$ this will return $[1]$, otherwise it will return some other value.

2. The variable is then assigned a label (if in an assignment clause) or is used in a computation (if in a compute clause).

The ABB functionality is implemented using a linear secret-sharing scheme which uses MACs to provide security with abort against malicious adversaries.

The secret-sharing scheme has two global MAC keys, Δ^{CB} which is held by CB, and Δ^{OB} which is held by OB. Each variable is additively secret-shared between OB and CB. That is, given a variable x_i, OB holds a share $\mathbf{x}_i^{\mathsf{OB}}$ and CB holds a share $\mathbf{x}_i^{\mathsf{CB}}$ such that $\mathbf{x}_i^{\mathsf{OB}} + \mathbf{x}_i^{\mathsf{CB}} = x_i$. The MAC keys are used to prevent a party from modifying their share. For each variable, x_i, OB is given a local MAC key, $\boldsymbol{\delta}_i^{x,\mathsf{OB}}$, and a local MAC $\mathbf{M}_i^{x,\mathsf{OB}}$. Likewise CB holds $\boldsymbol{\delta}_i^{x,\mathsf{CB}}$ and $\mathbf{M}_i^{x,\mathsf{CB}}$. These satisfy the conditions:

$$\mathbf{M}_i^{x,\mathsf{CB}} = \mathbf{x}_i^{\mathsf{CB}} \cdot \Delta^{\mathsf{OB}} - \boldsymbol{\delta}_i^{x,\mathsf{OB}}$$
$$\mathbf{M}_i^{x,\mathsf{OB}} = \mathbf{x}_i^{\mathsf{OB}} \cdot \Delta^{\mathsf{CB}} - \boldsymbol{\delta}_i^{x,\mathsf{CB}}$$

The ABB implementation makes use of two types of correlated randomness. The primary form of correlated randomness that we need is *autheticated Beaver triples*. These are used to implement the command Multiply. They consist of secret-sharings of three values, x, y and z, in which x and y are chosen uniformly at random from $\mathbb{Z}_p$ and $z = x \cdot y$. The second form of randomness required, used by CB-Input and OB-Input, is simply a secret-sharing of a value chosen uniformly at random from $\mathbb{Z}_p$. Since authenticated Beaver triples trivially provide two instances of this, we will only discuss how to generate Beaver triples.

Functionality 4.5. $\mathcal{F}_{\mathsf{Triple}}(p, \eta)$**: Authenticated Beaver Triples**

This functionality is accessed by parties CB and OB, with the (large) prime p and number of triples η as parameters. All operations in this functionality are modulo p. Upon receiving $(\mathsf{sid}, \texttt{gen-triples})$ from both CB and OB such that sid is fresh, sample:

- CB's shares $\mathbf{x}^{\mathsf{CB}}, \mathbf{y}^{\mathsf{CB}}, \mathbf{z}^{\mathsf{CB}}$ and MAC keys $\Delta^{\mathsf{CB}}, \boldsymbol{\delta}^{x,\mathsf{CB}}, \boldsymbol{\delta}^{y,\mathsf{CB}}, \boldsymbol{\delta}^{z,\mathsf{CB}}$
- OB's shares $\mathbf{x}^{\mathsf{OB}}, \mathbf{y}^{\mathsf{OB}}, \mathbf{z}^{\mathsf{OB}}$ and MAC keys $\Delta^{\mathsf{OB}}, \boldsymbol{\delta}^{x,\mathsf{OB}}, \boldsymbol{\delta}^{y,\mathsf{OB}}, \boldsymbol{\delta}^{z,\mathsf{OB}}$
- MACs held by CB: $\mathbf{M}^{x,\mathsf{CB}}, \mathbf{M}^{y,\mathsf{CB}}, \mathbf{M}^{z,\mathsf{CB}}$
- MACs held by OB: $\mathbf{M}^{x,\mathsf{OB}}, \mathbf{M}^{y,\mathsf{OB}}, \mathbf{M}^{z,\mathsf{OB}}$

such that the secrets maintain the following relationship for each $i \in [\eta]$:

$$(\mathbf{x}_i^{\mathsf{CB}} + \mathbf{x}_i^{\mathsf{OB}})(\mathbf{y}_i^{\mathsf{CB}} + \mathbf{y}_i^{\mathsf{OB}}) = \mathbf{z}_i^{\mathsf{CB}} + \mathbf{z}_i^{\mathsf{OB}}$$

and the MACs maintain the following relationship for each $i \in [\eta]$:

$$\begin{aligned} \mathbf{M}_i^{x,\mathsf{CB}} &= \mathbf{x}_i^{\mathsf{CB}} \cdot \Delta^{\mathsf{OB}} - \boldsymbol{\delta}_i^{x,\mathsf{OB}} & \qquad \mathbf{M}_i^{x,\mathsf{OB}} &= \mathbf{x}_i^{\mathsf{OB}} \cdot \Delta^{\mathsf{CB}} - \boldsymbol{\delta}_i^{x,\mathsf{CB}} \\ \mathbf{M}_i^{y,\mathsf{CB}} &= \mathbf{y}_i^{\mathsf{CB}} \cdot \Delta^{\mathsf{OB}} - \boldsymbol{\delta}_i^{y,\mathsf{OB}} & \qquad \mathbf{M}_i^{y,\mathsf{OB}} &= \mathbf{y}_i^{\mathsf{OB}} \cdot \Delta^{\mathsf{CB}} - \boldsymbol{\delta}_i^{y,\mathsf{CB}} \\ \mathbf{M}_i^{z,\mathsf{CB}} &= \mathbf{z}_i^{\mathsf{CB}} \cdot \Delta^{\mathsf{OB}} - \boldsymbol{\delta}_i^{z,\mathsf{OB}} & \qquad \mathbf{M}_i^{z,\mathsf{OB}} &= \mathbf{z}_i^{\mathsf{OB}} \cdot \Delta^{\mathsf{CB}} - \boldsymbol{\delta}_i^{z,\mathsf{CB}} \end{aligned}$$

Send $(\mathsf{sid}, \texttt{shares}, \mathbf{x}^{\mathsf{CB}}, \mathbf{y}^{\mathsf{CB}}, \mathbf{z}^{\mathsf{CB}}, \mathbf{M}^{x,\mathsf{CB}}, \mathbf{M}^{y,\mathsf{CB}}, \mathbf{M}^{z,\mathsf{CB}}, \Delta^{\mathsf{CB}}, \boldsymbol{\delta}^{x,\mathsf{CB}}, \boldsymbol{\delta}^{y,\mathsf{CB}}, \boldsymbol{\delta}^{z,\mathsf{CB}})$ to CB, and $(\mathsf{sid}, \texttt{shares}, \mathbf{x}^{\mathsf{OB}}, \mathbf{y}^{\mathsf{OB}}, \mathbf{z}^{\mathsf{OB}}, \mathbf{M}^{x,\mathsf{OB}}, \mathbf{M}^{y,\mathsf{OB}}, \mathbf{M}^{z,\mathsf{OB}}, \Delta^{\mathsf{OB}}, \boldsymbol{\delta}^{x,\mathsf{OB}}, \boldsymbol{\delta}^{y,\mathsf{OB}}, \boldsymbol{\delta}^{z,\mathsf{OB}})$ to OB.

In case either party is corrupt, it may submit its own shares rather than receiving them as output, in which case the other party's shares are computed to ensure the above equalities.

Given access to this functionality, we can implement the ABB as follows.

Protocol 4.6. $\pi_{\mathsf{ABB}}(p, \eta_I, \eta_M)$**: Arithmetic Black Box**

These subprotocols are run by parties CB and OB, with the (large) prime p as a parameter—all operations are modulo p.

We use the shorthand $[x]$ to denote the secret sharing of x along with its MAC. In particular, $[x]$ is characterized by $(x^{\mathsf{CB}}, M^{x,\mathsf{CB}}, \Delta^{\mathsf{CB}}, \delta^{x,\mathsf{CB}})$ held by CB, and $(x^{\mathsf{OB}}, M^{x,\mathsf{OB}}, \Delta^{\mathsf{OB}}, \delta^{x,\mathsf{OB}})$ held by OB. The protocol context will ensure that these values are related as

$$M^{x,\mathsf{CB}} = x^{\mathsf{CB}} \cdot \Delta^{\mathsf{OB}} - \delta^{x,\mathsf{OB}} \qquad M^{x,\mathsf{OB}} = x^{\mathsf{OB}} \cdot \Delta^{\mathsf{CB}} - \delta^{x,\mathsf{CB}}$$

Each subprotocol implicitly accepts two private arguments, corresponding to the private inputs of CB and OB respectively, however we will use the shorthand as described above for simplicity. Each subroutine (with the exception of **Open**) outputs shares for each party of the same format as the input—with the global MAC keys Δ^{OB},Δ^{CB} omitted as they never change.

For some subprotocols, we specify shorthand that we invoke in order to substitute them in place.

Init(p, η_I, η_M)**:**

1. Set the modulus used for all operations to p.
2. Let $\eta = \lceil \frac{\eta_I}{2} \rceil + \eta_M$.
3. Set $([x_i], [y_i], [z_i])_{1 \le i \le \eta} = \mathcal{F}_{\mathsf{Triple}}(p, \eta)$
4. Store $[Input_i] = [x_i]$ for $1 \le i \le \lceil \frac{\eta_I}{2} \rceil$.
5. Store $[Input_{i+\lceil \frac{\eta_I}{2} \rceil}] = [y_i]$ for $1 \le i \le \lfloor \frac{\eta_I}{2} \rfloor$.
6. Store $[Mult_i] = ([x_{i+\lceil \frac{\eta_I}{2} \rceil}], [y_{i+\lceil \frac{\eta_I}{2} \rceil}], [z_{i+\lceil \frac{\eta_I}{2} \rceil}])$ for $1 \le i \le \eta_M$.

OB**-Input**(y)**:**

1. Let $[x]$ be the first entry of $[Input\ ]$ which has not yet been used
2. OB obtains $x =$ **Open-to-**OB$([x])$
3. OB sends $d = y - x$ to CB.
4. Parties execute and output **Add-Const**$([x], d)$

CB**-Input**(y)**:**

1. Let $[x]$ be the first entry of $[Input]$ which has not yet been used
2. CB obtains $x = \textbf{Open-to-}\mathsf{CB}([x])$
3. CB sends $d = y - x$ to OB.
4. Parties execute and output $\textbf{Add-Const}([x], d)$

Open-to-CB $([x])$**:**

1. OB sends $(x^{\mathsf{OB}}, M^{x,\mathsf{OB}})$ to CB
2. CB validates $M^{x,\mathsf{OB}} = x^{\mathsf{OB}} \cdot \Delta^{\mathsf{CB}} - \delta^{x,\mathsf{CB}}$
3. CB outputs $x = x^{\mathsf{OB}} + x^{\mathsf{CB}}$

Open-to-OB $([x])$**:**

1. CB sends $(x^{\mathsf{CB}}, M^{x,\mathsf{CB}})$ to OB.
2. OB validates $M^{x,\mathsf{CB}} = x^{\mathsf{CB}} \cdot \Delta^{\mathsf{OB}} - \delta^{x,\mathsf{OB}}$
3. OB outputs $x = x^{\mathsf{OB}} + x^{\mathsf{CB}}$

Open$([x])$**:**

1. OB outputs the result of $\textbf{Open-to-}\mathsf{OB}([x])$
2. CB outputs the result of $\textbf{Open-to-}\mathsf{CB}([x])$

Add-Const$([x], c)$**:** Shorthand $[x] + c$.

1. OB outputs its share $(x^{\mathsf{OB}} + c,\ M^{x,\mathsf{OB}},\ \delta^{x,\mathsf{OB}})$
2. CB outputs its share $(x^{\mathsf{CB}},\ M^{x,\mathsf{CB}},\ \delta^{x,\mathsf{CB}} + c \cdot \Delta^{\mathsf{CB}})$

Mult-Const$([x], c)$**:** Shorthand $c \cdot [x]$.

1. OB outputs its share $(x^{\mathsf{OB}} \cdot c,\ M^{x,\mathsf{OB}} \cdot c,\ \delta^{x,\mathsf{OB}} \cdot c)$
2. CB outputs its share $(x^{\mathsf{CB}} \cdot c,\ M^{x,\mathsf{CB}} \cdot c,\ \delta^{x,\mathsf{CB}} \cdot c)$

Add$([x], [y])$**:** Shorthand $[x] + [y]$.

1. OB outputs its share $(x^{\mathsf{OB}} + y^{\mathsf{OB}},\ M^{x,\mathsf{OB}} + M^{y,\mathsf{OB}},\ \delta^{x,\mathsf{OB}} + \delta^{y,\mathsf{OB}})$
2. CB outputs its share $(x^{\mathsf{CB}} + y^{\mathsf{CB}},\ M^{x,\mathsf{CB}} + M^{y,\mathsf{CB}},\ \delta^{x,\mathsf{CB}} + \delta^{y,\mathsf{CB}})$

Multiply$([x], [y])$**:** Shorthand $[x] \cdot [y]$.

1. Let $([\hat{x}], [\hat{y}], [\hat{z}])$ be the first entry from $[Mult]$ which has not yet been used.

2. Parties set $[d] = [x] - [\hat{x}]$ and $[e] = [y] - [\hat{y}]$

3. Then they compute $d =$**Open**$([d])$ and $e =$**Open**$([e])$

4. Output $e \cdot [x] + d \cdot [y] - d \cdot e + [\hat{z}]$

Random():

1. OB picks a random $r_{\mathsf{OB}} \leftarrow \mathbb{Z}_p$.

2. CB picks a random $r_{\mathsf{CB}} \leftarrow \mathbb{Z}_p$.

3. OB-Input(r_{OB})

4. CB-Input(r_{CB})

5. Output $[r_{\mathsf{OB}}] + [r_{\mathsf{CB}}]$

TestBit$([x])$: Parties output $[b] = 1 - ([x] \cdot (1 - [x]))$

The protocol that realizes $\mathcal{F}_{\mathsf{Triple}}$ makes use of Vector Oblivious Linear Evaluation (VOLE) VOLE is an arithmetic generalization of OT, which is captured in the following functionality.

Functionality 4.7. $\mathcal{F}_{\mathsf{VOLE}}(p, \eta_V)$: Vector Oblivious Linear Evaluation

This functionality is accessed by parties CB and OB, with the (large) prime p and number of instances η_V as parameters. All operations in this functionality are modulo p. Upon receiving $(\mathsf{sid}, \texttt{gen-vole}, b)$ from both CB and OB such that sid is fresh and $b \in \{0, 1\}$, sample a secret $a \leftarrow \mathbb{Z}_p$ and three vectors $\mathbf{b}, \mathbf{c}, \mathbf{d} \in \mathbb{Z}_p^{\eta}$ subject to:

$$\mathbf{c}_i + \mathbf{d}_i = a \cdot \mathbf{b}_i \qquad \forall i \in [\eta_V]$$

Prepare messages

$$\mathsf{outp}_0 = (\mathsf{sid}, \texttt{vole-outp}, \mathbf{c}, a) \text{ and } \mathsf{outp}_1 = (\mathsf{sid}, \texttt{vole-outp}, \mathbf{b}, \mathbf{d})$$

and send outp_b to CB, and outp_{1-b} to OB.

In case either party is corrupt, it may submit its own shares rather than receiving them as output, in which case the other party's shares are computed to ensure the above equalities.

The above functionality can be realized by standard techniques based on Oblivious Transfer [Gil99, KOS16, DKLs18, HMRT22] or Homomorphic Encryption [Gil99, CGG+20, CCL+21]. We opt for the most recent OT based instantiation of Doerner et al. [DKLs24], due to its low comptutational footprint, and optimized communication.

Triple generation will also make use of a randomness sampling functionality $\mathcal{F}_{\mathsf{Rand}}$, which can be realized through a standard commit-and-release coin tossing

protocol.

Functionality 4.8. $\mathcal{F}_{\mathsf{Rand}}(p)$**: Public Randomness**

Upon receiving $(\mathsf{sid}, \texttt{sample})$ from both CB and OB such that sid is fresh, sample $\rho \leftarrow \mathbb{Z}_p$, and send $(\mathsf{sid}, \texttt{random-value}, \rho)$ to both parties.

Given $\mathcal{F}_{\mathsf{VOLE}}$ and $\mathcal{F}_{\mathsf{Rand}}$, it is possible to generated authenticated Beaver triples using the protocol π_{Triple}, presented below. This uses standard MPC techniques of triple sacrifice and VOLE-based MACs [DO10, DPSZ12, KOS16].

Protocol 4.9. $\pi_{\mathsf{Triple}}(p, \eta)$**: Authenticated Beaver Triples**

This protocol is run by parties CB and OB, with the (large) prime p and number of triples η as parameters. All operations in this protocol are modulo p.

Generate Triples:

1. For each $i \in [4\eta]$, CB and OB send $(\mathsf{sid}|0|i, \texttt{gen-vole}, 0)$ to $\mathcal{F}_{\mathsf{VOLE}}(p, 1)$ and receive as private outputs $(\mathsf{sid}|0|i, \texttt{vole-outp}, \mathbf{v}_i^{\mathsf{CB}}, \mathbf{w}_i^{\mathsf{CB}})$ and $(\mathsf{sid}|0|i, \texttt{vole-outp}, \mathbf{w}_i^{\mathsf{OB}}, \mathbf{v}_i^{\mathsf{OB}})$ respectively.

2. For each $i \in [2\eta]$, CB and OB respectively define:
$$\begin{aligned} \mathbf{x}_i^{\mathsf{CB}} &= \mathbf{w}_i^{\mathsf{CB}} & \mathbf{y}_i^{\mathsf{CB}} &= \mathbf{w}_{2\eta+i}^{\mathsf{CB}} & \mathbf{z}_i^{\mathsf{CB}} &= \mathbf{v}_i^{\mathsf{CB}} + \mathbf{v}_{2\eta+i}^{\mathsf{CB}} + \mathbf{w}_i^{\mathsf{CB}} \cdot \mathbf{w}_{2\eta+i}^{\mathsf{CB}} \\ \mathbf{x}_i^{\mathsf{OB}} &= \mathbf{w}_{2\eta+i}^{\mathsf{OB}} & \mathbf{y}_i^{\mathsf{OB}} &= \mathbf{w}_i^{\mathsf{OB}} & \mathbf{z}_i^{\mathsf{OB}} &= \mathbf{v}_i^{\mathsf{OB}} + \mathbf{v}_{2\eta+i}^{\mathsf{OB}} + \mathbf{w}_i^{\mathsf{OB}} \cdot \mathbf{w}_{2\eta+i}^{\mathsf{OB}} \end{aligned}$$

3. CB and OB send $(\mathsf{sid}|1, \texttt{gen-vole}, 0)$ to $\mathcal{F}_{\mathsf{VOLE}}(p, 6\eta)$ and receive as private outputs $(\mathsf{sid}|1, \texttt{vole-outp}, \mathbf{d}^{\mathsf{CB}}, \Delta^{\mathsf{CB}})$ and $(\mathsf{sid}|1, \texttt{vole-outp}, \mathbf{p}^{\mathsf{OB}}, \mathbf{M}^{\mathsf{OB}})$ respectively.

4. CB and OB send $(\mathsf{sid}|2, \texttt{gen-vole}, 1)$ to $\mathcal{F}_{\mathsf{VOLE}}(p, 6\eta)$ and receive as private outputs $(\mathsf{sid}|2, \texttt{vole-outp}, \mathbf{p}^{\mathsf{CB}}, \mathbf{M}^{\mathsf{CB}})$ and $(\mathsf{sid}|2, \texttt{vole-outp}, \mathbf{d}^{\mathsf{OB}}, \Delta^{\mathsf{OB}})$ respectively.

5. CB and OB respectively define $\boldsymbol{\gamma}^{\mathsf{CB}}$ and $\boldsymbol{\gamma}^{\mathsf{OB}}$ such that
$$\boldsymbol{\gamma}_i^{\mathsf{CB}} = \begin{cases} \mathbf{x}_i^{\mathsf{CB}} - \mathbf{p}_i^{\mathsf{CB}} & \text{if } i \in [1, 2\eta] \\ \mathbf{y}_{i-2\eta}^{\mathsf{CB}} - \mathbf{p}_i^{\mathsf{CB}} & \text{if } i \in [2\eta+1, 4\eta] \\ \mathbf{z}_{i-4\eta}^{\mathsf{CB}} - \mathbf{p}_i^{\mathsf{CB}} & \text{if } i \in [4\eta+1, 6\eta] \end{cases}$$
$$\boldsymbol{\gamma}_i^{\mathsf{OB}} = \begin{cases} \mathbf{x}_i^{\mathsf{OB}} - \mathbf{p}_i^{\mathsf{OB}} & \text{if } i \in [1, 2\eta] \\ \mathbf{y}_{i-2\eta}^{\mathsf{OB}} - \mathbf{p}_i^{\mathsf{OB}} & \text{if } i \in [2\eta+1, 4\eta] \\ \mathbf{z}_{i-4\eta}^{\mathsf{OB}} - \mathbf{p}_i^{\mathsf{OB}} & \text{if } i \in [4\eta+1, 6\eta] \end{cases}$$
and exchange these values.

6. For each $i \in [2\eta]$, CB and OB respectively define

$$\begin{aligned} \boldsymbol{\delta}_i^{x,\mathsf{CB}} &= \mathbf{d}_i^{\mathsf{CB}} + \Delta^{\mathsf{CB}} \cdot \boldsymbol{\gamma}_i^{\mathsf{OB}} & \boldsymbol{\delta}_i^{x,\mathsf{OB}} &= \mathbf{d}_i^{\mathsf{OB}} + \Delta^{\mathsf{OB}} \cdot \boldsymbol{\gamma}_i^{\mathsf{CB}} \\ \boldsymbol{\delta}_i^{y,\mathsf{CB}} &= \mathbf{d}_{2\eta+i}^{\mathsf{CB}} + \Delta^{\mathsf{CB}} \cdot \boldsymbol{\gamma}_{2\eta+i}^{\mathsf{OB}} & \boldsymbol{\delta}_i^{y,\mathsf{OB}} &= \mathbf{d}_{2\eta+i}^{\mathsf{OB}} + \Delta^{\mathsf{OB}} \cdot \boldsymbol{\gamma}_{2\eta+i}^{\mathsf{CB}} \\ \boldsymbol{\delta}_i^{z,\mathsf{CB}} &= \mathbf{d}_{4\eta+i}^{\mathsf{CB}} + \Delta^{\mathsf{CB}} \cdot \boldsymbol{\gamma}_{4\eta+i}^{\mathsf{OB}} & \boldsymbol{\delta}_i^{z,\mathsf{OB}} &= \mathbf{d}_{4\eta+i}^{\mathsf{OB}} + \Delta^{\mathsf{OB}} \cdot \boldsymbol{\gamma}_{4\eta+i}^{\mathsf{CB}} \end{aligned}$$

   Note at this point that CB and OB jointly hold 2η secret shared triples $[\mathbf{x}], [\mathbf{y}], [\mathbf{z}]$, where the ith triple $[\mathbf{x}_i], [\mathbf{y}_i], [\mathbf{z}_i]$ is characterized by

Value	CB's share	OB's share
$[\mathbf{x}_i]$	$(\mathbf{x}_i^{\mathsf{CB}}, \mathbf{M}_i^{x,\mathsf{CB}}, \Delta^{\mathsf{CB}}, \boldsymbol{\delta}_i^{x,\mathsf{CB}})$	$(\mathbf{x}_i^{\mathsf{OB}}, \mathbf{M}_i^{x,\mathsf{OB}}, \Delta^{\mathsf{OB}}, \boldsymbol{\delta}_i^{x,\mathsf{OB}})$
$[\mathbf{y}_i]$	$(\mathbf{y}_i^{\mathsf{CB}}, \mathbf{M}_i^{y,\mathsf{CB}}, \Delta^{\mathsf{CB}}, \boldsymbol{\delta}_i^{y,\mathsf{CB}})$	$(\mathbf{y}_i^{\mathsf{OB}}, \mathbf{M}_i^{y,\mathsf{OB}}, \Delta^{\mathsf{OB}}, \boldsymbol{\delta}_i^{y,\mathsf{OB}})$
$[\mathbf{z}_i]$	$(\mathbf{z}_i^{\mathsf{CB}}, \mathbf{M}_i^{z,\mathsf{CB}}, \Delta^{\mathsf{CB}}, \boldsymbol{\delta}_i^{z,\mathsf{CB}})$	$(\mathbf{z}_i^{\mathsf{OB}}, \mathbf{M}_i^{z,\mathsf{OB}}, \Delta^{\mathsf{OB}}, \boldsymbol{\delta}_i^{z,\mathsf{OB}})$

   such that $\mathbf{M}_i^{x,\mathsf{CB}} = \mathbf{M}_i^{\mathsf{CB}}, \mathbf{M}_i^{y,\mathsf{CB}} = \mathbf{M}_{2\eta+i}^{\mathsf{CB}}, \mathbf{M}_i^{z,\mathsf{CB}} = \mathbf{M}_{4\eta+i}^{\mathsf{CB}}$ and $\mathbf{M}_i^{x,\mathsf{OB}} = \mathbf{M}_i^{\mathsf{OB}}, \mathbf{M}_i^{y,\mathsf{OB}} = \mathbf{M}_{2\eta+i}^{\mathsf{OB}}, \mathbf{M}_i^{z,\mathsf{OB}} = \mathbf{M}_{4\eta+i}^{\mathsf{OB}}$. We will use shorthand for operations involving $[\mathbf{x}_i], [\mathbf{y}_i], [\mathbf{z}_i]$ going forward.

7. CB and OB invoke $\mathcal{F}_{\mathsf{Rand}}$ by sending $(\mathsf{sid}, \mathtt{sample})$ and receive $(\mathsf{sid}, \mathtt{random\text{-}value}, \rho)$ in response.

8. For each $i \in [\eta]$, CB and OB jointly do the following:

   (a) Compute $[\hat{\mathbf{z}}_i]$ as

   $$[\hat{\mathbf{z}}_i] = \mathbf{Multiply}([\mathbf{x}_i], [\mathbf{y}_i],\ \rho \cdot [\mathbf{x}_{\eta+i}],\ \rho \cdot [\mathbf{y}_{\eta+i}],\ \rho^2 \cdot [\mathbf{z}_{\eta+i}])$$

   that is, using $(\rho \cdot [\mathbf{x}_{\eta+i}],\ \rho \cdot [\mathbf{y}_{\eta+i}],\ \rho^2 \cdot [\mathbf{z}_{\eta+i}])$ in place of $([\hat{x}], [\hat{y}], [\hat{z}])$ as the sacrificed triple.

   (b) Compute and verify $\mathbf{Open}([\mathbf{z}_i] - [\hat{\mathbf{z}}_i]) \stackrel{?}{=} 0$

9. If all of the above checks pass, CB and OB output their respective shares of $([\mathbf{x}_i], [\mathbf{y}_i], [\mathbf{z}_i])_{i \in [\eta]}$.

   Note that half of the triples, i.e. $([\mathbf{x}_i], [\mathbf{y}_i], [\mathbf{z}_i])_{i \in [\eta+1, 2\eta]}$ are discarded.

Theorem 4.10. *Protocol π_{Triple} realizes $\mathcal{F}_{\mathsf{Triple}}$ in the $\mathcal{F}_{\mathsf{VOLE}}$,$\mathcal{F}_{\mathsf{Rand}}$ hybrid model.*

Proof. The protocol π_{Triple} is a reproduction of the protocols of Damgård and Orlandi [DO10] with the difference being that the commitment scheme is instantiated by means of $\mathcal{F}_{\mathsf{VOLE}}$ as in Bendlin et al. [BDOZ11]. We refer the reader to those works for the full proofs of the relevant components. □

4.2 Main Protocol

We first formalize the problem by means of an ideal functionality $\mathcal{F}_{\mathsf{PCFM}}$. The functionality is invoked by two parties, the *central bank*, denoted CB, and the *originating bank*, denoted OB.

Functionality 4.11. $\mathcal{F}_{\mathsf{PCFM}}(1^\lambda, L)$**: Private Capital Flow Management**

This functionality is accessed by parties CB and OB, and the capital flows list size, and the bit length of each entry B are fixed parameters. Upon receiving $(\mathsf{sid}, \texttt{capital-flows}, \mathbf{Y}, \mathbf{Z})$ from CB and $(\mathsf{sid}, \texttt{company-limit}, Y, X)$ from OB such that sid is fresh, X is a B-bit integer, and $\mathbf{Z}$ is a list of B-bit integers of the correct size, return (sid, b) to OB, where b is a bit such that $b = 1$ iff $X + \mathbf{Z}_Y < L$. If $Y \notin \mathbf{Y}$, the protocol aborts.

We now construct a two-party protocol π_{PCFM} to realize the above functionality in Protocol 4.13. This protocol requires a sub-protocol GreaterThan that securely compares two integers. We use the protocol of Garay et al. [GSV07], which is presented in Protocol 4.12 below. Intuitively, it is based on the fact that given $X, Y \in \{0,1\}^\ell$, $X > Y$ in exactly two cases. The former is that the higher-order $\ell/2$ bits of X are greater than the higher-order $\ell/2$ bits of Y. The latter is that the higher-order $\ell/2$ bits of X are equal to the higher-order $\ell/2$ bits of Y and the lower-order $\ell/2$ bits of X are greater than the lower-order $\ell/2$ bits of Y. The algorithm computes this recursively, where $t_{i,j}$ indicates that $X_{i+j-1} \dots X_i > Y_{i+j-1} \dots Y_i$ and $z_{i,j}$ indicates that $X_{i+j-1} \dots X_i = Y_{i+j-1} \dots Y_i$. Note that X_0 represents the least significant bit of X. Protocol GreaterThan is viewed as implementing an extension to $\mathcal{F}_{\mathsf{ABB}}$, allowing the ABB to also perform comparisons.

Protocol 4.12. $\pi_{\mathsf{Compare}}(\{[X_i]\}_{i\in[0,l-1]}, \{[Y_i]\}_{i\in[0,l-1]})$ **: Comparison**

This protocol is parameterized by the parties P_1 and P_2, and the bit-length l. The protocol runs between the parties P_1 and P_2, of which any one may be *maliciously* corrupt. The input is two variables X and Y, both stored in the ABB. They are represented bit-wise, i.e. as $\{[X_i]\}_{i\in[0,l-1]}$ and $\{[Y_i]\}_{i\in[0,l-1]}$ respectively. The output of the protocol for both parties is a boolean value c, where $c = 1$ if $X > Y$, and is 0 otherwise. The output is stored in the ABB, from which it may be assigned to a new variable, or used in a larger computation.
This protocol assumes that the inputs have the same bit-length ℓ (which is assumed to be a power of 2) and that the bit-wise representation of the inputs indeed consists of bits (which can be guaranteed by using the TestBit functionality).

GreaterThan$(\{[X_i]\}_{i\in[0,l-1]}, \{[Y_i]\}_{i\in[0,l-1]})$**:**

1. Both parties do the following:
   (a) Compute $[X_iY_i] = [X_i] \cdot [Y_i]$ for each $i \in [0, \ell - 1]$.

(b) Compute

$$[t_{i,1}] = [X_i] - [X_i Y_i] \text{ for } i \in [0, \ell - 1]$$
$$[z_{i,1}] = 1 - [X_i] - [Y_i] + 2[X_i Y_i] \text{ for } i \in [1, \ell - 1]$$

2. In each subsequent round $j \in [1, \log_2(\ell)]$ both parties do the following:

   (a) For each $i \in [1, \ell - 1]$ such that $i = 0 \mod 2^j$, compute

   $$[z_{i,2^j}] = [z_{i+2^{j-1},2^{j-1}}] \cdot [z_{i,2^{j-1}}]$$

   (b) For each $i \in [0, \ell - 1]$ such that $i = 0 \mod 2^j$, compute

   $$[t_{i,2^j}] = [t_{i+2^{j-1},2^{j-1}}] + [z_{i+2^{j-1},2^{j-1}}] \cdot [t_{i,2^{j-1}}]$$

3. Parties output $[t_{0,\ell}]$

We now present the full protocol for Capital Flow Management. As previously described, this is achieved by CB encrypting each of its entries by adding mask Z^{CB}, OB obtaining the encryption of its desired entry, $\mathbf{Z}_Y^{\mathsf{OB}}$, and comparing $X + \mathbf{Z}_Y^{\mathsf{OB}}$ with $L + Z^{\mathsf{CB}}$. However, several things can go wrong. Firstly, OB may not use the encryption it obtained. To remedy this, each encrypted entry is also provided with a MAC and the protocol verifies that the MAC is valid. Secondly, parties may input incorrect bitwise representation to the ABB for the comparison protocol: OB may provide an incorrect value for $X + \mathbf{Z}_Y^{\mathsf{OB}}$, and CB may provide an incorrect value for $L + Z^{\mathsf{CB}}$. These are verified, first by ensuring that the stored bit-wise representations actually contain bits using the $\mathsf{TestBit}(\cdot)$ protocol, and second by converting from bitwise back to arithmetic representations and ensuring that the equalities still hold. Lastly, CB could provide invalid inputs to its list of encrypted inputs, either it could provide values where the MAC is invalid, or it could provide an encrypted input that corresponds to an encryption of a negative number. If these were to be checked in different steps and aborted immediately, CB would learn that Y existed in particular subsets of $\mathbf{Y}$. To alleviate this, the protocol reveals only a single output which indicates whether $(\mathbf{Z}_Y^{\mathsf{OB}}, \mathbf{M}_Y)$ is a valid pair for some Y in $\mathbf{Y}$, and aborts if it is not.

Protocol 4.13. $\pi_{\mathsf{PCFM}}(1^\lambda)$**: Private Capital Flow Management**

This protocol is parameterized by the central bank CB, the originating bank OB. It is also parameterized by a balance bit-length B, a statistical security parameter λ_{s}, and a prime $p > 2^{l-1} + 2^{B+1}$, where $l = B + \lambda_{\mathsf{s}} + 1$. For instance $B = 47$, $\lambda_{\mathsf{s}} = 80$ and $l = 128$. It uses $\mathcal{F}_{\mathsf{PSt}}$, and is therefore also parameterized by the elliptic curve $\mathcal{G} = (\mathbb{G}, G, q)$. The protocol runs between the parties CB and OB, of which any one may be *maliciously* corrupt. The private input of this protocol for CB is a *list of existing capital flows* $\mathbf{Z}$ for a range of companies $\mathbf{Y}$ and for OB is the *company* Y

and a proposed transaction amount X. The private output of the protocol for OB is a boolean value b, where $b = 1$ if $X + \mathbf{Z}_Y < L$, 0 otherwise. If $Y \notin \mathbf{Y}$, the protocol aborts.

This protocol functions in the $\mathcal{F}_{\mathsf{PSt}}$,$\mathcal{F}_{\mathsf{ABB}}$-hybrid model. We use the notation $[s]$ to denote a variable s is stored in $\mathcal{F}_{\mathsf{ABB}}$. We assume that L is a public value which is already stored in $\mathcal{F}_{\mathsf{ABB}}$ and $L < 2^B$.

Private Transfer:

1. On input $(\mathsf{sid}, \texttt{company-limit}, Y, X)$, party OB does the following:
   (a) Send $(\mathsf{sid}, \texttt{company}, Y)$ to $\mathcal{F}_{\mathsf{PSt}}$.
2. On input $(\mathsf{sid}, \texttt{capital-flows}, \mathbf{Y}, \mathbf{Z})$, party CB does the following:
   (a) Sample a random mask $Z^{\mathsf{CB}} \leftarrow \mathbb{Z}_{2^{\ell-1}}$, and generate encrypted list $\mathbf{Z}^{\mathsf{OB}}$ as follows:
   $$\mathbf{Z}^{\mathsf{OB}} = \{\mathbf{Z}_y + Z^{\mathsf{CB}}\}_{y \in \mathbf{Y}}$$
   (b) CB defines $W = L + Z^{\mathsf{CB}}$
   (c) Sample $\alpha \leftarrow \mathbb{Z}_p$, $\beta \leftarrow \mathbb{Z}_p$ and compute the MAC on each element of $\mathbf{Z}^{\mathsf{OB}}$ as follows:
   $$\mathbf{M} = \{\alpha \cdot \mathbf{Z}_y^{\mathsf{OB}} + \beta \bmod p\}_{y \in \mathbf{Y}}$$
   (d) Send $(\mathsf{sid}, \texttt{masked-capital-flows-list}, \mathbf{Y}, \mathbf{Z}^{\mathsf{OB}}, \mathbf{M})$ to $\mathcal{F}_{\mathsf{PSt}}$.
3. On receiving $(\mathsf{sid}, \mathbf{Z}_Y^{\mathsf{OB}}, \mathbf{M}_Y)$ from $\mathcal{F}_{\mathsf{PSt}}$, party OB does the following:
   (a) If $(\mathbf{Z}_Y^{\mathsf{OB}}, \mathbf{M}_Y) = (\perp, \perp)$, set $\mathbf{Z}_Y^{\mathsf{OB}} = 0^\ell$ and $\mathbf{M}_Y \leftarrow \mathbb{Z}_p$. (This occurs when $Y \notin \mathbf{Y}$, in which case this picks an invalid MAC, which will later cause an abort.)
   (b) OB defines $U = X + \mathbf{Z}_Y^{\mathsf{OB}}$.
4. Both parties initialize the ABB, and each inputs their private variables.
   (a) Initialize the ABB to an empty namespace: $\mathcal{F}_{\mathsf{ABB}}.\mathrm{Init}(p)$
   (b) CB computes the bit decomposition $(Z_{l-1}^{\mathsf{CB}}, \dots, Z_0^{\mathsf{CB}})$ and $(W_{l-1}, \dots, W_0)$ of Z^{CB} and W, respectively.
   (c) For each $i \in [0, l-1]$, run:
   $$\mathsf{CB}\text{-}\mathsf{Input}(Z_i^{\mathsf{CB}})$$
   $$\mathsf{CB}\text{-}\mathsf{Input}(W_i)$$

(d) CB inputs their MAC keys:

$$\mathsf{CB\text{-}Input}(\alpha)$$

$$\mathsf{CB\text{-}Input}(\beta)$$

(e) Run the following:

$$\mathsf{OB\text{-}Input}(\mathbf{M}_Y)$$

(f) OB computes the bit decomposition $(\mathbf{Z}^{\mathsf{OB}}_{Y,l-1}, \ldots, \mathbf{Z}^{\mathsf{OB}}_{Y,0})$, $(X_{B-1}, \ldots, X_0)$ and $(U_{l-1}, \ldots, U_0)$ of $\mathbf{Z}^{\mathsf{OB}}_Y$, X and U, respectively.

(g) For each $i \in [0, B-1]$, run:

$$\mathsf{OB\text{-}Input}(X_i)$$

(h) For each $i \in [0, l-1]$, run:

$$\mathsf{OB\text{-}Input}(\mathbf{Z}^{\mathsf{OB}}_{Y,i})$$

$$\mathsf{OB\text{-}Input}(U_i)$$

(i) If $\mathsf{Open}([Z^{\mathsf{CB}}_{l-1}]) \neq 0$, abort.

5. Each party $\mathsf{P} \in \{\mathsf{OB}, \mathsf{CB}\}$ does the following:

(a) For each $i \in [0, l-1]$, let:

$$[\mathsf{t}^i] = \mathsf{TestBit}([W_i])$$

$$[\mathsf{t}^{l+i}] = \mathsf{TestBit}([\mathbf{Z}^{\mathsf{OB}}_{Y,i}])$$

$$[\mathsf{t}^{2l+i}] = \mathsf{TestBit}([U_i])$$

$$[\mathsf{t}^{3l+i}] = \mathsf{TestBit}([Z^{\mathsf{CB}}_i])$$

(b) For each $i \in [0, B-1]$, let:

$$[\mathsf{t}^{4l+i}] = \mathsf{TestBit}([X_i])$$

(c) If $\exists i \in [0, 4l+B-1]$ such that $\mathsf{Open}([\mathsf{t}^i]) \neq 1$, abort. Otherwise compute the following:

$$[Z^{\mathsf{CB}}] = \sum_{i=0}^{i=l-1} 2^i [Z^{\mathsf{CB}}_i]$$

$$[W] = \sum_{i=0}^{i=l-1} 2^i [W_i]$$

$$[\mathbf{Z}^{\mathsf{OB}}_Y] = \sum_{i=0}^{i=l-1} 2^i [\mathbf{Z}^{\mathsf{OB}}_{Y,i}]$$

$$[X] = \sum_{i=0}^{i=B-1} 2^i [X_i]$$

$$[U] = \sum_{i=0}^{i=l-1} 2^i [U_i]$$

$$[V] = [U] - [X] - [\mathbf{Z}^{\mathsf{OB}}_Y]$$

$$[T] = [W] - [L] - [Z^{\mathsf{CB}}]$$

(d) If $\mathsf{Open}([V]) \neq 0$ or $\mathsf{Open}([T]) \neq 0$, abort. Otherwise let:

$$[c] = \mathsf{GreaterThan}(\{[Z^{\mathsf{CB}}_i]\}_{i\in[0,l-1]}, \{[\mathbf{Z}^{\mathsf{OB}}_{Y,i}]\}_{i\in[0,l-1]})$$

$$[b] = \mathsf{GreaterThan}(\{[W_i]\}_{i\in[0,l-1]}, \{[U_i]\}_{i\in[0,l-1]})$$

(e) Run the following:

$$[a] = \mathsf{Multiply}([\alpha], [\mathbf{Z}^{\mathsf{OB}}_Y])$$

$$[z] = \mathsf{Add}([a], [\beta]) - [\mathbf{M}_Y]$$

(f) Let:

$$[r] = \mathsf{Random}()$$

$$[f] = (1 - (1 - [c]) \cdot (1 - [z])) \cdot [r]$$

(g) If $\mathsf{Open}([f]) \neq 0$, abort. Otherwise output $\mathsf{Open}([b])$.

Theorem 4.14. *Protocol π_{PCFM} realizes functionality $\mathcal{F}_{\mathsf{PCFM}}$ in the $\mathcal{F}_{\mathsf{PSt}}$,$\mathcal{F}_{\mathsf{ABB}}$ hybrid model.*

Proof. (Sketch) We describe how the transcript of each party can be simulated upon corruption. Note that the parties do not send any messages directly to each other. Therefore, the transcript will consist solely of outputs which are provided to the parties by the functionalities $\mathcal{F}_{\mathsf{PSt}}$ and $\mathcal{F}_{\mathsf{ABB}}$.

CB **corrupted**

First, the simulator determines if the protocol will abort due to invalid inputs. If CB provides any W_i or Z^{CB}_i which is not a bit, this will be revealed, and the protocol aborts. The simulator then computes the appropriate outputs to TestBit for these values and records these in the transcript with the resulting abort. The opened values of TestBit for values provided by OB are all set to 1 in the transcript (as OB is honest so will provide bits).

If CB provides W and Z^{CB} such that $W \neq Z^{\mathsf{CB}} + L$, then the simulator calculates the value of T that would be obtained, and records this being opened

on the transcript and the resulting abort. In both cases the value of V that is opened is 0, as OB is honest.

Next, the simulator extracts CB's input to $\mathcal{F}_{\mathsf{PCFM}}$. It takes the value of Z^{CB} that CB inputs to $\mathcal{F}_{\mathsf{ABB}}$. It also takes the values of α and β which CB provides to $\mathcal{F}_{\mathsf{ABB}}$. Finally, it takes $\mathbf{Y}$ and $\mathbf{Z}^{\mathsf{OB}}$ as provided to $\mathcal{F}_{\mathsf{PSt}}$. For each $y \in \mathbf{Y}$, if $\mathbf{Z}_y^{\mathsf{OB}} - Z^{\mathsf{CB}} \geq 0$ and $\mathbf{M}_y = \alpha \mathbf{Z}_y^{\mathsf{OB}} + \beta$, it adds $(y, \mathbf{Z}_y^{\mathsf{OB}} - Z^{\mathsf{CB}})$ to (Y', Z'). It then provides (Y', Z') as the input to $\mathcal{F}_{\mathsf{PCFM}}$. In the case where $\mathcal{F}_{\mathsf{PCFM}}$ aborts, the simulator picks a random $f \leftarrow [1, p-1]$ and records in the transcript that this value was opened and an abort occurred. Note that this abort corresponds to 3 different possibilities in the protocol execution: that the queried Y was really not in $\mathbf{Y}$, that CB provided a value of $\mathbf{Z}_Y^{\mathsf{OB}}$ that, for the given mask, represented a negative value, and the case that $\mathbf{M}_Y$ was an invalid MAC. These 3 cases are intentionally indistinguishable, showing that if CB provides invalid inputs for cetain values from $\mathbf{Y}$, it is as if these were never included in its database. In the case where $\mathcal{F}_{\mathsf{PCFM}}$ does not abort, the value opened for f is 0 and the transcript records the same output as $\mathcal{F}_{\mathsf{PCFM}}$.

OB **corrupted**

First the simulator determines if the protocol will abort due to invalid inputs. If OB provides any U_i, $\mathbf{Z}_{Y,i}^{\mathsf{OB}}$ or X_i which are not bits, the simulator calculates the values that will be revealed by the $\mathsf{TestBit}$ protocol, adds these to the transcript, and records an abort. Not that all outputs of $\mathsf{TestBit}$ on bits provided by CB will be 1 as CB is honest.

If OB provides U, $\mathbf{Z}_Y^{\mathsf{OB}}$ and X such that $U \neq \mathbf{Z}_Y^{\mathsf{OB}} + X$, then the simulator calculates the value of V that would be obtained, and records this being opened on the transcript and the resulting abort. In both cases the value of T that is opened is 0 as CB is honest.

Next, the simulator extracts OB's input to $\mathcal{F}_{\mathsf{PCFM}}$. Y is set to the value provided by OB to $\mathcal{F}_{\mathsf{PSt}}$. X is set to the value that was input to $\mathcal{F}_{\mathsf{ABB}}$, as the bits X_i. If OB inputs to $\mathcal{F}_{\mathsf{ABB}}$ a different $\mathbf{Z}_Y^{\mathsf{OB}}$ or $\mathbf{M}_Y$ to the ones provided by $\mathcal{F}_{\mathsf{PSt}}$, the simulator picks a random $f \leftarrow [1, p-1]$ as the value opened for f and aborts at this point. Otherwise it records $f = 0$ as being opened and records the same output as $\mathcal{F}_{\mathsf{PCFM}}$.

□

References

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.

[BKM+20] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Private matching for compute. Cryptology ePrint Archive, Report 2020/599, 2020. https://eprint.iacr.org/2020/599.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CCL+21] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Bandwidth-efficient threshold EC-DSA revisited: Online/offline extensions, identifiable aborts, proactivity and adaptive security. Cryptology ePrint Archive, Report 2021/291, 2021. https://eprint.iacr.org/2021/291.

[CGG+20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1769–1787. ACM Press, November 2020.

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th ACM STOC*, pages 364–369. ACM Press, May 1986.

[DKLs18] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *2018 IEEE Symposium on Security and Privacy*, pages 980–997. IEEE Computer Society Press, May 2018.

[DKLs24] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Threshold ECDSA in three rounds. *IEEE Symposium on Security and Privacy*, 2024.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Heidelberg, August 2007.

[DO10] Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 558–576. Springer, Heidelberg, August 2010.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[Gil99] Niv Gilboa. Two party RSA key generation. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 116–129. Springer, Heidelberg, August 1999.

[GSV07] Juan A. Garay, Berry Schoenmakers, and José Villegas. Practical and secure solutions for integer comparison. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *PKC 2007*, volume 4450 of *LNCS*, pages 330–342. Springer, Heidelberg, April 2007.

[HFH99] Bernardo A. Huberman, Matt Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, page 7886, New York, NY, USA, 1999. Association for Computing Machinery.

[HMRT22] Iftach Haitner, Nikolaos Makriyannis, Samuel Ranellucci, and Eliad Tsfadia. Highly efficient OT-based multiplication protocols. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 180–209. Springer, Heidelberg, May / June 2022.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[JKKX16] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *2016 IEEE European Symposium on Security and Privacy*, pages 276–291, 2016.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Heidelberg, August 2015.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.

[Roy22] Lawrence Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 657–687. Springer, Heidelberg, August 2022.