Securely Computing One-Sided Matching Markets

James Hsin-Yu Chiang[1⋆], Ivan Damgård[1⋆⋆], Claudio Orlandi[1] ⋆⋆⋆, Mahak Pancholi[2] †, and Mark Simkin[3] ‡

[1] Aarhus University
[2] IMDEA Software Institute
[3] Independent Researcher

Abstract. Top trading cycles (TTC) is a famous algorithm for trading indivisible goods between a set of agents such that all agents are as happy as possible about the outcome. In this paper, we present a protocol for executing TTC in a privacy preserving way. To the best of our knowledge, it is the first of its kind. As a technical contribution of independent interest, we suggest a new algorithm for determining all nodes in a functional graph that are on a cycle. The algorithm is particularly well suited for secure implementation in that it requires no branching and no random memory access. Finally, we report on a prototype implementation of the protocol based on somewhat homomorphic encryption.

1 Introduction

Barter economies, where agents directly exchange goods amongst each other, are one of the oldest forms of commerce. While historically barter was restricted to physical goods or services, nowadays barter is gaining popularity as a form of commerce in the context of cryptocurrencies and decentralized finance. In this digital realm, the goods that are being traded are so-called tokens, which may represent assets, intellectual property rights, equities, bonds, or services.

In our work, we consider one of the most archetypical forms of barter, known as one-sided matching markets, introduced by Shapley and Scarf [SS74]. Here, we have n agents, each holding one (indivisible) good as well as an ordered preference list over the n goods. The agents are willing to engage in a joint trading protocol and each agent would like to get their most preferred good. The protocol should ensure that all agents are as "happy" as they could be, once trading has finished, i.e., no subset of agents can still perform trades amongst each other that would leave all agents in the subset better off. Shapley and Scarf [SS74] showed that, for arbitrary preference lists, a sequence of trades that makes everybody happy

⋆ jachiang@cs.au.dk
⋆⋆ ivan@cs.au.dk
⋆⋆⋆ orlandi@cs.au.dk
† mahak.pancholi@imdea.org
‡ mark@univariate.org

always exists and presented an algorithm, known as the *top trading cycles* (TTC) algorithm[4], that efficiently computes the necessary trades.

The one-sided matching problem is not only a very intuitive game-theoretic problem, but its instances can be found in several real-world scenarios, with the TTC algorithm being used for effective solutions in many cases; for example, assigning an optimal allocation of schools for pupils [APR09, Abd11, APRS05, Jac19], public housing allocation [Tha16], and mutual housing exchange, etc.

The TTC algorithm has several attractive game-theoretic properties. Roth and Postlewaite [RP77] showed that, if the preferences of each agent are strict[5], the algorithm finds the unique allocation of goods to agents. Later, Roth [Rot82b] showed that all agents are incentivized to be truthful, i.e. that no agent can obtain a better good by being dishonest about their claimed preference list.

The appealing game-theoretic features of the TTC algorithm make it a useful tool for extending the potential of barter trading in the context of trading digital tokens in decentralized finance. Let's say that a particular token represents a particular service, and a user in possession of a token x would like to exchange it for y, and if not y, then for z. Traditionally, the bulk of digital tokens that are exchanged are limited to exchange between a pair of users, and only for preference over one token at a time. With TTC, users can post their preferences in the beginning as a ranked preference list of all offers from all other participating users [6], and receive the optimal exchange solution.

Unfortunately, a naive implementation of TTC would require all involved agents to publicly reveal their preference lists, which in turn would also reveal who is obtaining which good. When transaction privacy is required, this is not a viable solution. However, there has been no prior work studying secure-TTC.

A first (naive) approach to implement a privacy preserving version of TTC would be to simply convert the TTC algorithm to a circuit in a straightforward way and evaluate it using a generic Multiparty Computation (MPC) engine. This would result in a rather inefficient solution, typically requiring $O(n^2 \log(n))$ rounds of communication for n agents. One problem with the straightforward approach is the large number of random memory accesses required, which is notoriously difficult to implement in secure computation. We discuss this in more detail in Sec. 1.3. A main contribution of this work, discussed in more detail in the following, is to come up with a new equivalent formulation of the TTC algorithm that is much better suited for secure computation because it is algebraic in nature and requires no memory accesses.

1.1 Our Contribution

In this work, we present a protocol that allows n agents to efficiently compute all the desirable trades in a given one-sided matching market, without revealing any unnecessary private information. Our protocol hides each agent's preference

[4] Attributed to David Gale.
[5] In the sense that no agent likes two goods equally.
[6] A user can reject offers from other parties, by ordering its own offer above theirs.

list and agents only learn about the trades they are personally involved in. Surprisingly, our work is the first to address this question, as far as we are aware of. Our protocol does not require any random memory accesses or any expensive branching operations and for this reason it integrates well with secure computation frameworks for arithmetic circuits.

As a technical building block that may be of independent interest, we construct a simple and (asymptotically) highly efficient protocol for determining the nodes that are part of a cycle on a hidden functional[7] graph (Sec 3).

Our new secure TTC protocol can be realized from any MPC framework that offers basic arithmetic in a prime field of cardinality larger than the number of parties. We prove our protocol to be UC-secure in the semi-honest, dishonest majority setting (Sec 4.1) and we show that each protocol subtask induces a multiplicative depth that is *logarithmic* in the number of participants.[8]

We experimentally evaluate our TTC protocol by building a prototype implementation[9] based on somewhat homomorphic encryption (SHE), also known as leveled-HE. Although our current implementation can still be optimized in several ways, it already shows that the approach has the potential in practice, e.g., the entire protocol can be done for 25 users in a few minutes. Such practical runtimes are achieved by heavily exploiting purpose-built and general SIMD (Same-Instruction Multiple-Data) techniques for SHE schemes that natively support SIMD operations. Our implementation is based on the OpenFHE [ABBB+22] framework and the BGV cryptosystem [BGV12]. We provide benchmarks for various parameter settings (Sec 5).

Our construction can be implemented based on any secure computation framework offering basic arithmetic in finite fields. So an obvious question is whether it would be more efficient to use a secret-sharing MPC protocol, like SPDZ [DPSZ12], rather than SHE. However, secret-sharing based MPC incurs a large number of communication rounds, which becomes the main bottleneck as soon as the number of parties or the round trip time of the network is large enough. While our objective in this work is not to compare secret-sharing and FHE in general and for all parameter ranges, we provide a discussion on the two implementation methods in Appx. H, targeted at our setting. We conclude that secret-sharing based MPC will be slower as soon as the network roundtrip time is large enough (40ms in our example setting).

Upgrading to malicious security can be done using standard techniques in a relatively straightforward way. Although the resulting protocol would less efficient, we expect that the overhead would not necessarily be prohibitive. We discuss this extension in more detail in Appendix I.

1.2 Related Work

In the following, we discuss research domains closely related to our work.

[7] A directed graph is said to be functional, if all vertices have out-degree at most one.

[8] In comparison to a naive implementation that would require a multiplicative depth linear in the number of parties.

[9] Source code has been uploaded here.

Matching Algorithms. Beyond one-sided matching markets, many other types of matching problems have been studied in the literature. These include: The stable marriage problem of Gale and Shapley [GS62], with its privacy-preserving variants presented in [Gol06, FGM07, KS14, ZWR+16, DEs16, MPA+23]. The housing allocation problem of Hylland and Zeckhauser [HZ79]. The kidney exchange problem of Roth, Sönmez, and Ünver [RSÜ04], with the privacy-preserving versions being recently proposed in [BHK+22, BHP+22, BMW22, BMW23]. From a technical perspective, the ideas for computing stable marriages or performing kidney exchange privately do not appear to be useful for solving the problem considered in this work; the former is incomparable, while the latter is a more restricted setting. On the other hand, we observe that the protocols in our work can easily be adapted to solve the housing allocation problem as well. In Appx A we provide a more detailed discussion about related matching algorithms.

Secure Graph Computations. Looking ahead, our protocol is based on the original TTC algorithm which repeatedly interprets agents as graph vertices, preferences as edges, and attempts to identify agents that are part of a graph cycle. While there are many works [BSA13, ACM+13, WRD+17, MKNK15, AFO+21] on secure computation of graph algorithms, such as determining the shortest path between two nodes in a graph, for performing depth/breadth first search, and computing the maximum flow of a graph, these tools, however, do not seem amenable to securely and efficiently determining *which* agents are part of a cycle. The difficulty of our task is best illustrated by considering the Floyd's famous cycle finding algorithm for functional graphs. While this algorithm has a simple condition for checking whether a cycle exists, it is not obvious how to modify it, such that it allows for efficiently *listing* all vertices that are part of the cycles.

1.3 Technical Overview

To understand the ideas behind our approach, let us first review the TTC algorithm itself and see why naively using secure computation techniques is unlikely to yield an efficient protocol.

The Top Trading Cycles Algorithm. Recall that we have n agents, each holding a private preference list, sorting all n goods from most to least desirable. In the TTC algorithm, in each round, every agent points at the agent with the good they desire the most. Viewing the agents as vertices and who they point to as edges, we get a functional graph with n vertices and n edges. Such graphs always have at least one cycle. Any agent that is part of a cycle will trade their good, i.e. they will receive the good they desire and they will give their good to whoever is pointing at them. All agents that were involved in trades leave the procedure and all remaining agents repeat this process by now pointing to their most preferred good among those that are still available. Eventually the algorithm terminates with all agents having performed trades, possibly with themselves.

Demands to a privacy-preserving solution. We will aim to construct a protocol that allows each party to only learn the trade she is involved in. In particular, the protocol must not leak the round in which her trade was decided, nor ask her to post new preference every round depending on the current availability. Instead, all parties must supply a complete preference list up front, and then the protocol must securely update the preferences between cycle finding steps "inside" the secure computation.

A Naive Approach. Let us focus on just one round of the TTC algorithm, where agents would like to determine whether they are part of a cycle or not. In the following, implicitly assume that all computations are done either on secret shared or encrypted values, depending on the precise secure computation framework that is used.

First, every agent publishes the index of the agent with their most preferred good, among those that are still present (ignoring for now how precisely this would even be done). Interpret all those pointers as an array A of length n. Now the agents will jointly perform n steps as follows: Initially, each agent i has an associated value $v_i = 1$ and is located at vertex i. At each step, each agent i looks up the successor of the vertex they are currently at in A and move to that vertex. Let j be this vertex, then agent i updates $v_i := v_i \cdot (i - j)$. If an agent i is part of a cycle, then it will have returned to their initial node within n steps at least once and thus $v_i = 0$ after n steps.

While this solution does indeed allow each agent to determine whether they are part of a cycle, it also requires each agent to perform n memory lookups in array A *securely*. When implemented via a naive circuit, this would require one linear scan of A per access per agent. A more intelligent approach is to use secure computation protocols for RAM programs [GKK+12], which can perform efficient memory accesses (as low as $O(\log(n))$) access per RAM access). Many protocols for secure RAM computation have already been proposed [ZWR+16, Ds17, KY18, BKKO20, HV21, VHG23, BPRS23, SVG23, NFO24], but those are either restricted to a constant number of parties, require an honest majority among the parties, or are significantly less efficient than circuit-based secure computation protocols. Consequently, it would be desirable to have a protocol that does not require any random memory accesses and can be expressed nicely as circuit.

In any case, for both naive RAM and circuit approaches, one round of TTC would require $O(n)$ sequential memory accesses, amounting to a multiplicative depth of $O(n)$ per round. As opposed to this, our solution only requires $O(\log(n))$ multiplicative depth per round of TTC.

Our Solution. The main idea underlying our approach is to view the graph through its adjacency matrix and to exploit certain structural properties of these matrices that are specific to functional graphs. The adjacency matrix M of a graph with n vertices is an $n \times n$ matrix, where entry $(i, j) \in \{1, \ldots, n\} \times \{1, \ldots, n\}$ is one, if there is an edge from node i to node j and zero otherwise. It is well known that for any $k \in \mathbb{N}$, the entry (i, j) in M^k equals the number of walks from node

i to node j of length k. Intuitively, this would already allow for checking whether vertex i is on a cycle by computing all powers $M^1, \ldots, M^n$ of the adjacency matrix M and checking whether in any of them, there is a non-zero entry at (i, i), i.e. whether there is a walk of some length $\ell \in \{1, \ldots, n\}$ from vertex i to itself. This would work, but would require n separate matrix-matrix multiplications. In this work, we build upon this basic idea, but reduce the number of matrix-matrix multiplications to $\log(n)$.

What we prove in this work, is that computing $u = M^n \cdot \mathbf{1}$, where $\mathbf{1}$ is the column vector of length n with all entries being one, allows for determining the vertices that are on cycles. Concretely, we prove that for each $i \in \{1, \ldots, n\}$, the i-th value in u is non-zero if and only if vertex i is on a cycle, provided the underlying graph is a functional graph. Note that computing M^n can be done with $\log(n)$ matrix-matrix multiplications via repeated squaring. Furthermore, note that that this approach does not require any random memory accesses at all and is purely algebraic in nature.

While efficiently determining which vertices are on a cycle in a given functional graph is one of the more difficult steps, there are several other technical difficulties that need to be overcome, e.g., the secure updating of preferences, alluded to above. We will highlight those and our corresponding solutions in detail in the technical sections of this work.

2 Preliminaries

Notation. We denote scalars as x, vectors as $\boldsymbol{v}$, matrices as A, and A^T as the transpose of A. We write $\mathbf{1}$ to denote the vector of length n, where all entries are 1. We write $\boldsymbol{v} \cdot \boldsymbol{w}$ to denote hadamard product, i.e. element-wise multiplication of vectors. We write $v_1 \to v_2$ to denote a directed edge from vertex v_1 to v_2. For a value a, we write $[a]$ to denote the encryption of value a.

2.1 Secure Multiparty Computation

We prove security in the UC framework [Can01, CLOS02] with semi-honest and static corruptions, and $\mathcal{F}'$-hybrid setting. The security requirement is captured by showing indistinguishability between the real-world and ideal-world experiments, where in the ideal-world all of the computation is done via an ideal functionality $\mathcal{F}$. For a brief summary and formal definition, see Appx. B.

2.2 Ideal Functionality: Top Trading Cycles

We describe the algorithm of Shapely and Scarf [SS74], as already discussed in Sec 1.3, in the UC functionality F_{TTC} in Fig. 1.

2.3 Leveled Homomorphic Encryption

To instantiate our TTC protocol we will use a leveled homomorphic encryption scheme (leveled-HE). The standard definition is reproduced in Appx. C.

F_{TTC}

F_{TTC} is an n party functionality and runs with clients $\{C_1, \ldots, C_n\}$.

Input: For $i \in \{1, \ldots, n\}$, receive preference list $\boldsymbol{x}^{(i)}$ from client C_i.
Top Trading Cycle: Initialize the set of available clients $\mathcal{C} := \{C_1 \ldots, C_n\}$.
While $\mathcal{C} \neq \emptyset$, do:

1. For each $C_i \in \mathcal{C}$, let $\mathsf{top}_i := C_j$ be its first preference such that $C_j \in \mathcal{C}$.
2. Build a graph $\mathcal{G} := (V, E)$, where $V := \mathcal{C}$, and $(v_i \to v_j) \in E$ if $\mathsf{top}_i := C_j$.
3. Find all cycles in $\mathcal{G}$.
4. For each $v_i \in V$, if v_i lies on a cycle such that $v_i \to v_j$, store $(\mathsf{On\ Cycle}, i, j)$, and remove C_i from $\mathcal{C}$.

Output: Output $(\mathsf{On\ Cycle}, i, j)$ to client C_i where tuple $(\mathsf{On\ Cycle}, i, j)$ is stored internally.

Fig. 1: Functionality for Secure Top Trading Cycles

Specifically, our protocol is implemented with the BGV cryptosystem [BGV12], which offers ciphertext slots over which "same instruction multiple data" (SIMD) parallelism can be exploited without additional overhead (Sec. 4.2).

2.4 Ideal Functionality: Arithmetic Black Box

This functionality, called F_{ABB} (Appx. D, Fig. 8) provides an interface for doing a series of basic arithmetic operations on secret values in a secure manner, and to open the final output towards a particular participant. Trivially, by design, no information about the intermediate values is leaked to participants.

At a high level, F_{ABB} receives commands from two types of computing devices: from clients it receives an input vector of fixed length ℓ (via INPUT command), and then it allows servers $S_1, \ldots, S_m$ to securely perform element-wise additions (via ADD) and multiplications (via MULT) by making a single call to the functionality, i.e., ℓ parallel additions or multiplications can be computed at the cost of a single call. Additionally, the servers can securely *cycle* vector elements (via ROT) by any number of slots and direction; this operation is called *rotation*. At last, the participants can open the final output towards a particular client (via OPEN).

In the client-server setting, we can UC-realize F_{ABB} using an leveled-HE encryption scheme in the $F_{\mathsf{KeyGenDec}}$ hybrid (Appx. D, Fig. 9). $F_{\mathsf{KeyGenDec}}$ allows servers and clients to obtain a public-key for the leveled-HE scheme where the associated secret-key is stored inside the functionality. Given the public-key, clients can encrypt secret inputs and send them to a server S_1, who then evaluates homomorphic operations over ciphertexts locally with Eval algorithm of the leveled-HE scheme.

We give the details of this realization in Appx. D, The security of this realization can be formally stated in the following theorem. The proof is similar to that in [DPSZ12], but for completeness we present the main ideas in Appx. D.

Theorem 1. *Let* $\mathsf{LHE} := (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ *be a* leveled-HE *scheme that is correct,* IND-CPA *secure, and is circuit private. Then* F_{ABB} *can be UC-realized by a protocol with* LFHE *in the* $F_{\mathsf{KeyGenDec}}$*-hybrid against any static, passive adversary corrupting up to* $m-1$ *servers and* $n-1$ *clients.*

3 Basic Algorithms

As part of our overall solution, we will require solutions for two smaller problems. First, for each node in the graph, we need to decide if the node is on a cycle or not. Second, after a round of cycle finding is over, we need to compute the new preferences of all clients for the next round. Both of our solutions for these sub-tasks are designed to be easy to implement within secure computation.

Cycle finding. For determining the parties that are on a cycle, we will exploit properties of the adjacency matrix of the corresponding functional graph. Let matrix M be the adjacency matrix, where entry $M_{i,j}$ is 1, if the graph has a vertex from node i to node j and 0 otherwise. It is well known that $M^k_{i,j}$ is the number of paths of length k from node i to node j. As explained earlier, a simplistic way to exploit this would be to compute $M^2, M^3, ..$ and for each node i test, if any value $M^k_{i,i}$ is non-zero. If we, however, exploit the fact that we have a *functional graph*, i.e., all out-degrees are 1, then we obtain a significantly more efficient solution: node i is on a cycle if the i-th index in the vector $M^n\mathbf{1}$ is non-zero.

Specifically, we show the following technical lemma (proof in Appx. E):

Lemma 1. *Let* M *be the adjacency matrix of a functional graph with* n *nodes, let* $\mathbf{1}$ *be a column vector where all entries are 1 and let* $\boldsymbol{u} = M^n\mathbf{1}$*. Then* $\boldsymbol{u}_i$ *is non-zero if and only if node* i *is on a cycle. Moreover,* $\boldsymbol{u}_i$ *is a non-negative integer and* $\boldsymbol{u}_i \leq n$*.*

Using this lemma, we can securely decide whether nodes are on cycles by computing $M^n\mathbf{1}$ and checking which entries are 0. We can do this efficiently with leveled-HE, since matrix multiplications can be done using one layer of parallel multiplications (and some additions), thus matrix exponentiation to power n can be done using $\log n$ multiplicative depth via standard repeated squaring.

Preference Computation. Here, we assume that the preference list of client C_i is given as an $n \times n$ permutation matrix N_i, such that multiplying a vector by N_i will reorder the input entries in order of preference. We assume that the list of available goods is given as a vector $\boldsymbol{h}$, where $\boldsymbol{h}_j = 1$ if good number j is currently available, and 0 otherwise. As we shall see, such a list is readily available, once a cycle finding stage is done.

Our goal here is to compute the adjacency matrix of the graph for the next cycle finding stage. That is, for each C_i, we want to compute vector $\boldsymbol{w}^{(i)}$ where $\boldsymbol{w}^{(i)}_j = 1$ if house j is the one party i prefers among the available houses, and 0 otherwise. Viewing these individual vectors as a single matrix, we obtain our desired adjacency matrix. We can compute these vectors as follows:

1. Let vector $\boldsymbol{a}^{(i)} = N_i \boldsymbol{h}$.
2. Compute the vector $\boldsymbol{b}^{(i)}$ as follows:
   For $j = 1$ to n, set $\boldsymbol{b}_j^{(i)} = \boldsymbol{a}_j^{(i)} \prod_{k<j}(1 - \boldsymbol{a}_k^{(i)})$.
3. Let $\boldsymbol{w}^{(i)} = N_i^{-1} \boldsymbol{b}^{(i)}$.

For correctness, note that in vector $\boldsymbol{a}^{(i)}$, the first 1 corresponds to the good C_i prefers the most among the available ones. The formula for computing $\boldsymbol{b}^{(i)}$ preserves the first 1 in $\boldsymbol{a}^i$ but will zero out everything else. Thus, as desired, $\boldsymbol{w}^{(i)}$ will contain a 1 in the position of the good C_i prefers, and 0s elsewhere.

The reason for using this specific way of computing the adjacency matrix is that it can be done in depth $\log n$ and that we can exploit known algorithms for parallel prefix computation, such that the second step above only requires $O(n)$ multiplication, while still being logarithmic depth.

4 TTC Protocols: Generic and SIMD Optimized

In this section, we present two protocols for computing the TTC algorithm securely. In Sec 4.1, we present our main and generic protocol (Π_{TTC}) using basic secure arithmetic operations provided by F_{ABB}. We chose to first explain a simple version of our protocol, without any optimizations, to expose the central ideas of our TCC protocol. Π_{TTC} is generic since, to implement this, F_{ABB} can be realized by different techniques such as an MPC, or computation over ciphertexts.

Later, in Sec 4.2, we improve upon this by exploiting SIMD operations offered by leveled-HE schemes. Our optimised TTC protocol ($\Pi_{\mathsf{TTC-SIMD}}$) utilizes the SIMD interface of F_{ABB} to significantly reduce the number of multiplications by up to a factor of n^2, where n is the number of clients. This allows us to demonstrate practical runtimes with our implementation in Sec. 5.

4.1 Generic TTC Protocol

In this section, we present our main protocol (Π_{TTC} in Fig. 2) for computing the top trading cycles algorithm securely. We state correctness in Lem. 2, security in Thm. 2, and state the multiplicative depth of Π_{TTC} in Thm. 3 (proofs are stated in Appx. F).

Client-Server Model. We consider computations in the standard client-server model with n clients and m servers. The clients own the input and initialize F_{ABB} with it. The servers then do the computation over these inputs by repeatedly calling F_{ABB}, without any further involvement of the clients. Only towards the end, when the output is computed, the clients are again involved to learn the output. In Appx. D, we show how to realize F_{ABB} using an leveled-HE scheme.

Overview of Protocol Π_{TTC}. The protocol description appears in Figure 2. It includes three sub-protocols: Exponentiate, NotEqualZero and PreserveLeadOne, explained below.

Input: The clients first input their preference over goods in form of permutation matrices (N_i), which order offered goods by their preference, and their transpose to F_{ABB}. The clients also initialize vectors $\boldsymbol{h}$ and $\boldsymbol{o}$, which record the availability of goods, and the allocated goods, respectively. Once the client inputs are stored in F_{ABB}, subsequent parts of the protocol are performed by the servers calling arithmetic operations on the values stored in F_{ABB}.

Subsequently, one round of TTC algorithm consists of three steps: (i) compute current preference matrix (M) according to the available clients and their available preference, (ii) compute cycles and identify parties on the cycles, and (iii) update goods' availability and clients' assignments. These steps are repeated n times, since the TTC algorithm requires n rounds of cycle finding.

Preference computation: The client's permutation matrix (N_i) is used to reorder $\boldsymbol{h}$ by preference (Sec. 3). Then, PreserveLeadOne is used to isolate the most preferred, available good. Finally, applying the transpose gives us the i'th row of the adjacency matrix M. Recall from Sec. 3, given input sequence $x_1, ..., x_n$, to preserve the first 1, we compute $y_j = x_j \prod_{k<j}(1 - x_k)$ for $j = 1$ to n. Here, note that $\prod_{k<j}(1 - x_k)$ for $j = n$ represents the sequence of all prefix multiplications on $(1 - x_1), ..., (1 - x_{n-1})$; prefix multiplication is known to be computable in $\lceil \log(n) \rceil$ multiplicative depth [HSJ86], and we illustrate an example of prefix multiplication in Fig. 11.

Compute cycles: Let M be the adjacency matrix obtained in the previous step. The servers evaluate Exponentiate to compute M^n. We apply square-and-multiply to compute matrix exponentiation; to obtain M^n, first compute $M^2, M^4, ...M^{2^k}$, where $k = \lfloor \log(n) \rfloor$. Given $(i_k, \cdots, i_0)$, the binary representation of n, we then multiply terms $M^{i_0} \cdot M^{2 \cdot i_1} \ldots \cdot M^{2^k \cdot i_k}$ in binary tree fashion with $\lceil \log(\lfloor \log_2(n) \rfloor) \rceil$ depth. The resultant matrix is multiplied by a n-dimensional vector of ones, obtaining vector $\boldsymbol{u}$ with non-zero values in the i'th position, if the i'th client is on a cycle (as shown in Lem. 1).

Next, we map each non-zero element of $\boldsymbol{u}$ to 1 with NotEqualZero so that the i-th index of $\boldsymbol{u}$ is 1 if the client C_i is on a cycle, and otherwise 0. To compute NotEqualZero, we can exploit the fact that values in $\boldsymbol{u}$ never exceed n (Lem. 1), and thus we simply compute $\mathsf{NotEqualZero}(u_i) := 1 - n! \cdot \prod_{j=1}^{j=n}(j - u_i)$, rather than exploiting Fermat's little theorem: $x \neq 0 \iff x^{p-1} = 1$ and $x = 0 \iff x^{p-1} = 0$. Multiplying n terms only requires $O(\log(n))$ multiplicative depth.

Update availability of goods and client's assignment: Lastly, after each cycle finding round, we update clients' output vector $\boldsymbol{o}$ and availability of goods $\boldsymbol{h}$. We recover the index of each client's preferred good from the current adjacency matrix (M) and assign it to the output, if the output is unassigned $\boldsymbol{o}_i = 0$. Finally, the availability of goods is computed from the output vector.

Lemma 2. *(Correctness) Π_{TTC} (Figure 2) implements the top trading cycle algorithm (specified in F_{TTC}).*

$\Pi_{\mathsf{TopTradingCycle}}$

Parties $\mathcal{C} = \{C_1, \ldots, C_n\}$ interact with F_{ABB} to initialize their inputs. Then, computing servers $\mathcal{S} = \{S_1, ..., S_m\}$ interact with hybrid functionality F_{ABB} exclusively to securely evaluate the top trading cycle algorithm. Values in F_{ABB} are denoted by $[\cdot]$ for which a unique object identifier (id) is known to all servers. Servers execute stateless subroutines $\mathsf{PreserveLeadOne}$, $\mathsf{Exponentiate}$, and $\mathsf{NotEqualZero}$ on values stored in F_{ABB}.

Input: On receiving a preference list $\boldsymbol{x}^{(i)} \in \mathbb{Z}_p^n$ as input, each client C_i locally computes the permutation matrix $N_i \in \mathbb{Z}_p^{n \times n}$ such that $\boldsymbol{x}^{(i)} = N_i \times (1, ..., n)^T$, and privately inputs N_i and N_i^{-1} to F_{ABB}. Servers jointly initialize availability $[\boldsymbol{h}] \leftarrow 0^n$ and output vectors $[\boldsymbol{o}] \leftarrow 1^n$ in F_{ABB}.

Cycle Finding: For round $r \in (1, ..., n)$, servers jointly perform the following.

1. Update adjacency matrix:
   (a) For $i \in (1, ..., n)$:
      i. Let $\left[\boldsymbol{a}^{(i)}\right] \leftarrow ([N_i]\,[\boldsymbol{h}])$
      ii. Let $\left[\boldsymbol{b}^{(i)}\right] \leftarrow \mathsf{PreserveLeadOne}(\left[\boldsymbol{a}^{(i)}\right])$
      iii. Let $\left[\boldsymbol{w}^{(i)}\right] \leftarrow \left[N_i^{-1}\right] \times \left[\boldsymbol{b}^{(i)}\right]$
   (b) Store matrix $[M] \in \mathbb{Z}^{n \times n}$ with i'th row $\left[\boldsymbol{w}^{(i)}\right]^T$
2. Compute cycles:
   (a) $[\boldsymbol{u}] \leftarrow \mathsf{Exponentiate}([M], n) \times 1^n$
   (b) $[\boldsymbol{u}_i] \leftarrow \mathsf{NotEqualZero}([\boldsymbol{u}_i])$ for $i \in n$.
3. Update assignments & availability:
   (a) For $i \in (1, ..., n)$:
      i. $[\boldsymbol{t}_i] \leftarrow \sum_{j \in [n]} j \cdot [M_{i,j}]$
      ii. $[\boldsymbol{o}_i] \leftarrow [\boldsymbol{t}_i] \cdot [\boldsymbol{u}_i] + [\boldsymbol{o}_i] \cdot (1 - [\boldsymbol{u}_i])$
   (b) If round $r \neq n$, $i \in (1, ..., n)$:
      i. $[\boldsymbol{h}_i] \leftarrow (1 - \mathsf{NotEqualZero}([\boldsymbol{o}_i]))$

Open assignments: Clients call F_{ABB} to privately open $[\boldsymbol{o}_i]$ to C_i for $i \in (1, ..., n)$.

Fig. 2: Top trading cycle in the F_{ABB}-hybrid model.

Theorem 2. *Π_{TTC} (Figure 2) UC-realizes F_{TTC} (Figure 1) in the F_{ABB}-hybrid model against a passive adversary corrupting up to $m - 1$ servers and $n - 1$ clients.*

Multiplicative depth of Π_{TTC}. We have highlighted the multiplicative depth of subroutines $\mathsf{Exponentiate}$, $\mathsf{Not\ Equal\ Zero}$ and $\mathsf{PreserveLeadOne}$. The following theorem and proof states that given n client inputs, Π_{TTC} incurs a maximum

	Mults/Rots in each cycle finding round			Mult. Depth
	(1) Adj Matrix	(2) Cycle Comp	(3) Avail Update	
Π_{TCC}	$O(n^3)$	$O(n^3 \log(n)$	$O(n^2)$	$O(\log(n))$
$\Pi_{\mathsf{TCC\text{-}SIMD}}$	$O(n^2)$	$O(n \log(n))$	$O(n \log(n))$	

Fig. 3: Asymptotic complexities of Π_{TCC} and $\Pi_{\mathsf{TCC\text{-}SIMD}}$

multiplicative depth of $O(\log(n))$, for a single round of TTC, and $O(n \log(n))$ overall. We refer to Appx. F for the proof for all theorems.

Theorem 3. *Protocol Π_{TTC} evaluated on n client preference lists privately input to hybrid functionality F_{ABB} incurs a maximum multiplicative depth of $O(n \log(n))$ on values output from F_{ABB}.*

4.2 TTC Protocol with SIMD Optimizations

Whilst Π_{TCC} can be evaluated in $O(\log(n))$ multiplicative depth per cycle finding round, the concrete complexity remains high (Fig. 3). For example, NotEqualZero and PreserveLeadOne all require secure multiplication of n values in F_{ABB}. These operations are each repeated n times for each client, incurring $O(n^2)$ multiplications per cycle finding round. In each round, building the adjacency matrix and matrix exponentiation incur $O(n^3)$ and $O(n^3 \log(n))$ total multiplications, resp.

Our $\Pi_{\mathsf{TTC-SIMD}}$ in Fig. 6 exploits the full SIMD interface of F_{ABB} to reduce the total complexity of Π_{TTC} whilst retaining multiplicative depth of $O(\log(n))$, thereby enabling *practical runtimes* of secure TTC in our implementation (Sec.5). In $\Pi_{\mathsf{TTC-SIMD}}$, steps (1),(2), and (3) have improved asymptotic complexity over Π_{TTC} by up to a factor of n^2 (see comparison in Fig. 3). Here, it is important to also consider rotations as "expensive" operations as these incur runtimes in the same order of magnitude as multiplications when F_{ABB} is instantiated with leveled-HE (Fig. 15).

In the rest of the section we focus on an overview of $\Pi_{\mathsf{TTC\text{-}SIMD}}$, introducing additional sub-protocols needed to exploit SIMD operations effectively. Since, each sub-protocol retains the multiplicative depth of $O(\log(n))$ of $\Pi_{\mathsf{TTC\text{-}SIMD}}$, we only discuss the total number of operations needed. In Fig. 5, we provide the concrete number of addition, multiplication, and rotation operations needed for each of the three main steps in $\Pi_{\mathsf{TTC-SIMD}}$ (steps (1),(2),(3)). We refer to Appx. G for a more detailed description of $\Pi_{\mathsf{TTC\text{-}SIMD}}$. In particular, we highlight Fig. 14 which illustrates the cost for each SIMD-style sub-protocol; for these, we now provide a brief overview of how practical efficiencies can be achieved with SIMD optimizations.

$\mathsf{PrefixAdd}_{\mathsf{L/R}}$ and $\mathsf{PrefixMult}_{\mathsf{L/R}}$ protocols compute the sum/product of all prefixes of a vector with only $\log(n)$ additions/multiplications and rotations. Here, we only explain PrefixAdd for left (and right) directions; computation for PrefixMult is analogous. $\mathsf{PrefixAdd}_{\mathsf{L}}$ ($\mathsf{PrefixAdd}_{\mathsf{R}}$) outputs a vector where index i stores the sum of all input elements up to (starting from) index i.

We adapt parallel prefix arithmetic from [HSJ86] to the setting of SIMD operations in F_{ABB}. First, an n-length vector is padded with 0s (for $\mathsf{PrefixMult}$ we pad with 1s) to length $n' \geq 2^{k+1} - 1$, where $k = \lceil \log_2(n) \rceil$. For example, to compute $\mathsf{PrefixAdd}_{\mathsf{L}}$ for vector $[1, 2, \ldots, 8]$, we pad 0s to obtain $[1, 2, \ldots, 8, 0^7]$, and then proceed in levels. For level $i \in [0, \lceil \log_2(n) \rceil - 1]$, we rotate the intermediate vector to the *right* by 2^i slots (see left of Fig. 11). The unrotated vector from the preceding level is then added element-wise in SIMD-fashion with a single addition call to F_{ABB}. After $\lceil \log_2(n) \rceil$ levels, we obtain the resultant vector where each element holds the sum of all elements of the input vector with lower slot indices. This requires only $O(\log_2(n))$ total additions, and rotations. These prefix algorithms are essential to implement $\mathsf{InnerProd}$ and $\mathsf{PreserveLeadOne}$ in SIMD with concrete efficiency.

$\mathsf{InnerProd}$ is a task needed to update assignments and availability. The $\mathsf{InnerProd}$ over n-sized vectors $[\boldsymbol{v}]$, $[\boldsymbol{w}]$ can be computed with a single multiplication and evaluating the additive prefix over the result:

$$\mathsf{PrefixAdd}_{\mathsf{R}}([\boldsymbol{v}] \cdot [\boldsymbol{w}]) = \left[\sum_{i \in [m]} v_i w_i, \sum_{j \in [m-1]} v_j w_j, \ldots \right]$$

The inner product scalar is then located in the first slot position, and if necessary can be *extracted* by multiplying with a fresh encryption of $[1, 0, \ldots]$. This incurs a total multiplicative complexity of 1 (or 2 if extraction is required).

$\mathsf{PreserveLeadOne}$ is adapted for SIMD operations by padding the input vector $\boldsymbol{x}$ with 1s, and then executing SIMD operations on the following:

$$[\boldsymbol{x}, 1, \ldots, 1] \cdot \mathsf{rot}(\mathsf{PrefixMult}_{\mathsf{L}}([\mathbf{1}] - [\boldsymbol{x}, 1, \ldots, 1]), 1, \mathsf{right}) \tag{1}$$

Here, the padding with 1's maintains the correctness of the prefix multiplication following the rotation of vector elements. This incurs $\lceil \log_2(n) \rceil + 2$ multiplications, and $\lceil \log_2(n) \rceil + 1$ rotations.

$\mathsf{NotEqualZero}$ is a pure SIMD algorithm, that computes element-wise only; it does not call rotations in F_{ABB}. Each vector index represents a separate parallel execution. $\mathsf{NotEqualZero}$ for all elements in input vector $[\boldsymbol{v}]$, is evaluated as;

$$\mathsf{NotEqualZero}_n([\boldsymbol{v}]) = [\mathbf{1}] - [\boldsymbol{n!^{-1}}] \prod_{i \in [n]} ([\boldsymbol{i}] - [\boldsymbol{v}]) \tag{2}$$

Here, let $[\boldsymbol{i}]$ and $[\boldsymbol{n!^{-1}}]$ denote $[i, \ldots, i]$ and $[n!^{-1}, \ldots, n!^{-1}]$ with the same dimension as the input vector. By moving the factor -1 out of the product term and considering cases of even or odd n, we rewrite as follows to avoid negation by multiplication;

$$\mathsf{NotEqualZero}_n([\boldsymbol{x}]) = \begin{cases} [\mathbf{1}] + [\boldsymbol{-n!^{-1}}] \prod_{i \in [n]} ([\boldsymbol{x}] + [\boldsymbol{-i}]) & n \text{ even} \\ [\mathbf{1}] + [\boldsymbol{n!^{-1}}] \prod_{i \in [n]} ([\boldsymbol{x}] + [\boldsymbol{-i}]) & n \text{ odd} \end{cases} \tag{3}$$

NotEqualZero in SIMD-fashion and input range $[0, n]$ incurs $n+1$ multiplications and additions $\lceil \log_2(n+1) \rceil$.

SIMD Matrix operations: For matrix-matrix products, we implement the technique by Jiang et al. [JKLS18], which requires only $O(n)$ multiplications for multiplying two $n \times n$-matrices. For matrix-vector products, we implement Halevi and Shoup [HS14] which also exhibits $O(n)$ multiplicative complexity. These techniques encrypt entire matrices (or their diagonals) in a single ciphertext and are reproduced in Appx. G.3 for the readers convenience.

5 Implementation

We benchmark the local running times of the servers performing the leveled-HE computation, which ignores ciphertext refreshing. Here, running $\Pi_{\mathsf{TCC\text{-}SIMD}}$ for 5 clients takes 14 seconds, which increases to 2 minutes for 15 clients and 8 minutes for 25 clients. We provide estimates for ciphertext refreshing assuming specific network conditions, which turn out to be very small compared to the local computing time.

Benchmarking Computational Overheads. We illustrate the running times for varying numbers of clients. In addition to SIMD parallelization, we implement our protocol with hardware parallelization across threads on a single server.

Fig. 4 shows the total local runtime for varying number of clients. We emphasize that the TTC protocol runs an additional round for every additional client, whilst each individual round increases in complexity, which explains the increase in runtimes with increasing number of clients. In Fig. 5, the running times are depicted for each phase of a single TTC round (and different numbers of clients); whilst updating the adjacency matrix appears quasi-linear in number of clients with sufficient parallelisation, the matrix exponentiation in the cycle computation phase is not, suggesting observable bottlenecks in memory bandwidth.

We implement our $\Pi_{\mathsf{TTC-SIMD}}$ protocol with the OpenFHE [ABBB+22] library with AVX2 support, using their implementation of the BGV cryptosystem [BGV12]. The main parameters of ring-LWE variant of BGV are the plaintext modulus p, ciphertext modulus q and ciphertext ring dimension N. Large ciphertext moduli are required to accommodate the noise growth of ciphertexts during homomorphic operations.

OpenFHE exposes automated parameter generation to derive parameters for a given plaintext modulus which (1) achieve a desired multiplicative depth and (2) a level of standardized security (e.g. equivalent to 128 bits) [ACC+21]. We set $p = 65537$, a plaintext modulus recommended by library authors for general applications over integers. This modulus allows for homomorphic computations with a multiplicative depth of $\sim$11. Using this parameterization, we obtain up to 32768 plaintext slots in each ciphertext.

We assume that all computations are performed by three servers of which at most two are corrupt. We ran our experiments on a PC with an 48-core Intel CPU and 96 GB of RAM, and varying number of threads (Fig. 4). Our

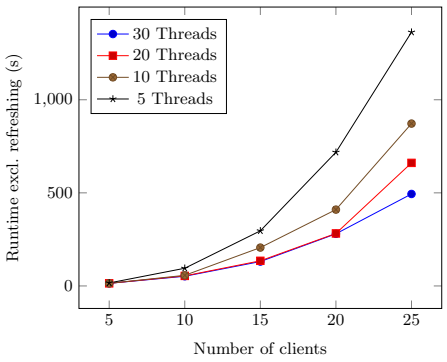

Fig. 4: TTC runtimes for different thread counts (excl. refreshing).

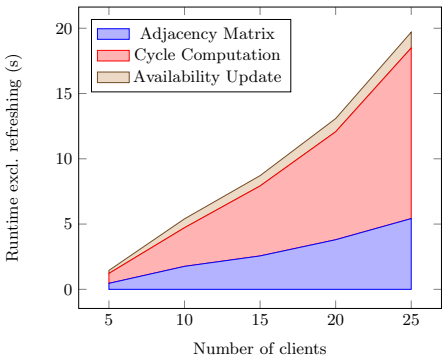

Fig. 5: TTC single round runtime (excl. refreshing) with 38 threads.

implementation of $\Pi_{\mathsf{TCC\text{-}SIMD}}$ already provides ciphertext level parallelisation; remaining protocol steps which can be parallelised at the hardware level are run with varying number of threads. We note that runtimes for n clients do not measurably improve beyond n threads, as single-thread performance and memory bandwidth become performance bottlenecks.

Refreshing intervals. Choosing the correct maximal depth for the encryption scheme we use, is a non-trivial task. It requires finding a good balance between computational overheads and round-trip latencies. Choosing a bigger maximal depth significantly affects the computational overhead of performing a single homomorphic multiplication, whereas a smaller maximal multiplicative depth, would increase the number of communication rounds. We model the communication overhead for refreshing ciphertexts assuming BGV parameters chosen for our implementation and various network conditions in Appx. H.

Comparison with secret-sharing based MPC. In SHE, the multiplicative depth between ciphertext refreshing is a fixed constant; in asymptotic terms, our communication overhead for re-freshing ciphertexts therefore grows linearly with the multiplicative depth, as is the case for secret-sharing based MPC. In concrete terms, however, we show in Appx. H that SHE is superior to secret-sharing based approaches for realistic, public network conditions, where the roundtrip latency exceeds ≈ 40 ms.

Conclusion. Our benchmarking results, using our new algorithmic insights, resolve one-sided matching markets within secure computation in practice. We note that deciding which trades to perform, may in practice often not be very time critical. For this reason, it may be acceptable for the protocols to run on the order of minutes.

An interesting future direction is to improve upon our implementation with optimizations for specific hardware architectures. In particular, we suspect memory bandwidth to be a key performance bottleneck, and given recent investments

to commercialize Application Specific Integrated Circuit (ASIC) designs for FHE evaluation promise to improve runtimes by an order of magnitude. Another direction of inquiry, which we leave for future works, is investigating whether (and for what parameters) an alternate implementation via secret-shared MPC techniques shows improvement.

References

ABBB+22. Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. Openfhe: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'22, pages 53–63, New York, NY, USA, 2022. Association for Computing Machinery.

Abd11. Atila Abdulkadiroglu. Generalized matching for school choice. *Unpublished paper, Duke University.[1311, 1312]*, 2011.

ACC+21. Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, et al. Homomorphic encryption standard. *Protecting privacy through homomorphic encryption*, pages 31–62, 2021. https://doi.org/10.1007/978-3-030-77287-1_2.

ACM+13. Abdelrahaman Aly, Edouard Cuvelier, Sophie Mawet, Olivier Pereira, and Mathieu Van Vyve. Securely solving simple combinatorial graph problems. In Ahmad-Reza Sadeghi, editor, *FC 2013: 17th International Conference on Financial Cryptography and Data Security*, volume 7859 of *Lecture Notes in Computer Science*, pages 239–257, Okinawa, Japan, April 1–5, 2013. Springer, Heidelberg, Germany.

AFO+21. Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 610–629, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

APR09. Atila Abdulkadiroğlu, Parag A Pathak, and Alvin E Roth. Strategy-proofness versus efficiency in matching with indifferences: Redesigning the nyc high school match. *American Economic Review*, 99(5):1954–1978, 2009.

APRS05. Atila Abdulkadiroğlu, Parag A Pathak, Alvin E Roth, and Tayfun Sönmez. The boston public school match. *American Economic Review*, 95(2):368–371, 2005.

AS98. Atila Abdulkadiroğlu and Tayfun Sönmez. Random serial dictatorship and the core from random endowments in house allocation problems. *Econometrica*, 66(3):689–701, 1998.

BGV12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325, Cambridge, MA, USA, January 8–10, 2012. Association for Computing Machinery.

BHK+22. Timm Birka, Kay Hamacher, Tobias Kussel, Helen Möllering, and Thomas Schneider. SPIKE: secure and private investigation of the kidney exchange problem. *BMC Medical Informatics Decision Making*, 22(1):253, 2022.

BHP+22. Malte Breuer, Pascal Hein, Leonardo Pompe, Ben Temme, Ulrike Meyer, and Susanne Wetzel. Solving the kidney exchange problem using privacy-preserving integer programming. In *19th Annual International Conference on Privacy, Security & Trust, PST 2022, Fredericton, NB, Canada, August 22-24, 2022*, pages 1–10. IEEE, 2022.

BKKO20. Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed ORAM. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20: 12th International Conference on Security in Communication Networks*, volume 12238 of *Lecture Notes in Computer Science*, pages 215–232, Amalfi, Italy, September 14–16, 2020. Springer, Heidelberg, Germany.

BMW22. Malte Breuer, Ulrike Meyer, and Susanne Wetzel. Privacy-preserving maximum matching on general graphs and its application to enable privacy-preserving kidney exchange. In Anupam Joshi, Maribel Fernández, and Rakesh M. Verma, editors, *CODASPY '22: Twelveth ACM Conference on Data and Application Security and Privacy, Baltimore, MD, USA, April 24 - 27, 2022*, pages 53–64. ACM, 2022.

BMW23. Malte Breuer, Ulrike Meyer, and Susanne Wetzel. Efficient privacy-preserving approximation of the kidney exchange problem. *CoRR*, abs/2302.13880, 2023.

BPRS23. Lennart Braun, Mahak Pancholi, Rahul Rachuri, and Mark Simkin. Ramen: Souper fast three-party computation for RAM programs. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. ACM, 2023.

BSA13. Marina Blanton, Aaron Steele, and Mehrdad Aliasgari. Data-oblivious graph algorithms for secure computation and outsourcing. In Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng, editors, *ASIACCS 13: 8th ACM Symposium on Information, Computer and Communications Security*, pages 207–218, Hangzhou, China, May 8–10, 2013. ACM Press.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

CLOS02. Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th Annual ACM Symposium on Theory of Computing*, pages 494–503, Montréal, Québec, Canada, May 19–21, 2002. ACM Press.

DEs16. Jack Doerner, David Evans, and abhi shelat. Secure stable matching at scale. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 1602–1613, Vienna, Austria, October 24–28, 2016. ACM Press.

DPR16. Ivan Damgård, Antigoni Polychroniadou, and Vanishree Rao. Adaptively secure multi-party computation from lwe (via equivocal fhe). In *Public-Key Cryptography–PKC 2016: 19th IACR International Conference on Practice*

and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part II 19, pages 208–233. Springer, 2016.

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, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

Ds17. Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 523–535, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

FGM07. Matthew K. Franklin, Mark Gondree, and Payman Mohassel. Improved efficiency for private stable matching. In Masayuki Abe, editor, *Topics in Cryptology – CT-RSA 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 163–177, San Francisco, CA, USA, February 5–9, 2007. Springer, Heidelberg, Germany.

Gen09. Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

GKK+12. S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012: 19th Conference on Computer and Communications Security*, pages 513–524, Raleigh, NC, USA, October 16–18, 2012. ACM Press.

Gol06. Philippe Golle. A private stable matching algorithm. In Giovanni Di Crescenzo and Avi Rubin, editors, *FC 2006: 10th International Conference on Financial Cryptography and Data Security*, volume 4107 of *Lecture Notes in Computer Science*, pages 65–80, Anguilla, British West Indies, February 27 – March 2, 2006. Springer, Heidelberg, Germany.

GS62. David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

HS14. Shai Halevi and Victor Shoup. Algorithms in HElib. Cryptology ePrint Archive, Report 2014/106, 2014. https://eprint.iacr.org/2014/106.

HSJ86. W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986. https://dl.acm.org/doi/pdf/10.1145/7902.7903.

HV21. Ariel Hamlin and Mayank Varia. Two-server distributed ORAM with sublinear computation and constant rounds. In Juan Garay, editor, *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 12711 of *Lecture Notes in Computer Science*, pages 499–527, Virtual Event, May 10–13, 2021. Springer, Heidelberg, Germany.

HZ79. Aanund Hylland and Richard Zeckhauser. The efficient allocation of individuals to positions. *Journal of Political economy*, 87(2):293–314, 1979.

Jac19. Rose Jacobs. Chicago booth. https://www.chicagobooth.edu/review/when-students-are-matched-schools-who-wins, 2019.

JKLS18. Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 1209–1222, 2018.

KS14. Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 506–525, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany.

KY18. Marcel Keller and Avishay Yanai. Efficient maliciously secure multiparty computation for RAM. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 91–124, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

MKNK15. Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. GRECS: Graph encryption for approximate shortest distance queries. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 504–517, Denver, CO, USA, October 12–16, 2015. ACM Press.

MPA+23. Arup Mondal, Priyam Panda, Shivam Agarwal, Abdelrahaman Aly, and Debayan Gupta. Fast and secure oblivious stable matching over arithmetic circuits. Cryptology ePrint Archive, Paper 2023/1789, 2023. https://eprint.iacr.org/2023/1789.

NFO24. Daniel Noble, Brett Hemenway Falk, and Rafail Ostrovsky. Metadoram: Breaking the log-overhead information theoretic barrier. *IACR Cryptol. ePrint Arch.*, 2024.

Rot82a. Alvin E. Roth. The economics of matching: Stability and incentives. *Mathematics of Operations Research*, 7(4):617–628, 1982.

Rot82b. Alvin E Roth. Incentive compatibility in a market with indivisible goods. *Economics Letters*, 9(2):127–132, 1982.

RP77. Alvin E Roth and Andrew Postlewaite. Weak versus strong domination in a market with indivisible goods. *Journal of Mathematical Economics*, 4(2):131–137, 1977.

RSÜ04. Alvin E Roth, Tayfun Sönmez, and M Utku Ünver. Kidney exchange. *The Quarterly Journal of Economics*, 119(2):457–488, 2004.

SS74. Lloyd Shapley and Herbert Scarf. On cores and indivisibility. *Journal of Mathematical Economics*, 1(1):23–37, 1974.

SVG23. Sajin Sasy, Adithya Vadapalli, and Ian Goldberg. PRAC: round-efficient 3-party MPC for dynamic data structures. *IACR Cryptol. ePrint Arch.*, 2023.

Tha16. Neil Thakral. The public-housing allocation problem. Technical report, Technical report, Harvard University, 2016.

VHG23. Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A bandwidth-efficient distributed ORAM for 2- and 3-party computation. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. USENIX Association, 2023.

WRD+17. Qian Wang, Kui Ren, Minxin Du, Qi Li, and Aziz Mohaisen. SecGDB: Graph encryption for exact shortest distance queries with efficient updates. In Aggelos Kiayias, editor, *FC 2017: 21st International Conference on Financial Cryptography and Data Security*, volume 10322 of *Lecture Notes in Computer Science*, pages 79–97, Sliema, Malta, April 3–7, 2017. Springer, Heidelberg, Germany.

ZWR+16. Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: Efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy*, pages 218–234, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press.

$\Pi_{\text{TTC-SIMD}}$

Clients $\mathcal{C} := \{C_1, \ldots, C_n\}$ and servers $\mathcal{S} := \{S_1, \ldots, S_m\}$ interact with functionality F_{ABB}. F_{ABB} is initialized to $\mathbb{Z}_p^{n^2}$, implying SIMD operations over vectors of length n^2. Inputs of length less than n^2 are padded with trailing 0s. Values stored in F_{ABB} are denoted by $[\cdot]$ for which a unique object identifier (id) is known to all parties.

Input: Each client C_i receives its preference list $\boldsymbol{x}^{(i)} \in \mathbb{Z}_p^n$ as input and performs the following:

1. Compute permutation matrix $N_i \in \mathbb{Z}^{n \times n}$ such that $\boldsymbol{x}^{(i)} = N_i \times [1, ..., n]^T$.
2. For $\ell \in [n]$, compute $\mathsf{diag}_\ell(N_i)$ and $\mathsf{diag}_\ell(N_i^{-1})$ (See eq. 7 for matrix diagonals).
3. Initialize $\mathsf{diag}_\ell(N_i), \mathsf{diag}_\ell(N_i^{-1})$ by calling F_{ABB} on INPUT.

All parties initialize $\boldsymbol{h}$ and $\boldsymbol{o}$, where $\boldsymbol{h} \leftarrow 1^{2n}$ and $\boldsymbol{o} \leftarrow 0^n$.

Cycle Finding: For round $r \in (1, ..., n)$ servers perform the following;

1. **Update adjacency matrix:**
   (a) For client index $i \in (1, ..., n)$:
      i. $\left[\boldsymbol{a}^{(i)}\right] \leftarrow \sum_{0 \le l < n} [\mathsf{diag}_l(N_i)] \cdot \mathsf{rot}([\boldsymbol{h}], l, \mathsf{left})$
      ii. $\left[\boldsymbol{a}^{(i)}\right] \leftarrow \left[\boldsymbol{a}^{(i)}\right] \cdot [(1^n, 0^n)] + [(0^n, 1^n)]$
      iii. Let $\left[\boldsymbol{b}^{(i)}\right] \leftarrow \mathsf{PreserveLeadOne}(\left[\boldsymbol{a}^{(i)}\right])$
      iv. $\left[\boldsymbol{b}^{(i)}\right] \leftarrow \left[\boldsymbol{b}^{(i)}\right] \cdot [(1^n, 0^n)]$
      v. $\left[\boldsymbol{b}^{(i)}\right] \leftarrow \left[\boldsymbol{b}^{(i)}\right] + \mathsf{rot}(\left[\boldsymbol{b}^{(i)}\right], n, \mathsf{right})$
      vi. $\left[\boldsymbol{w}^{(i)}\right] \leftarrow \sum_{0 \le l < n} [\mathsf{diag}_l(N_i^{-1})] \cdot \mathsf{rot}([\boldsymbol{b}^{(i)}], l, \mathsf{left})$
   (b) For $i \in [n]$, pack $[\boldsymbol{w}^{(i)}]$ as $[\boldsymbol{w}] := [\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)}, \ldots, \boldsymbol{w}^{(n)}]$ by running **Pack Vector** (Fig. 10).
2. **Compute cycles:**
   (a) Copy $[\boldsymbol{u}] \leftarrow [\boldsymbol{w}]$
   (b) For $r \in [1, ..., \lceil \log(n) \rceil]$: $[\boldsymbol{u}] \leftarrow \sum_{i \in [n-1]} \mathsf{colShift}_i(\mathsf{lin}([\boldsymbol{u}])) \cdot \mathsf{rowShift}_i(\mathsf{lin}'([\boldsymbol{u}]))$
   (c) Unpack $[\boldsymbol{u}]$, and for $l \in [n]$ initialize $\mathsf{diag}_l(\mathsf{Flat}^{-1}(\boldsymbol{u})) \in \mathbb{Z}^n$ by running **Pack Vector**[a].
   (d) $[\boldsymbol{u}] \leftarrow \mathsf{NotEqualZero}_n(\sum_{0 \le l < n} [\mathsf{diag}_l(U)])$
3. **Update assignments and availability:**
   (a) For $i \in [n]$, $[\boldsymbol{c}^{(i)}] \leftarrow \mathsf{innerProd}_{\mathsf{R}}([\boldsymbol{w}^{(i)}], [(1, 2, ..., n)])$
   (b) $[\boldsymbol{t}] \leftarrow \sum_{i \in [n]} \mathsf{rot}([(1, 0, ..., 0)] \cdot [\boldsymbol{c}^{(i)}], i - 1, \mathsf{right})$
   (c) $[\boldsymbol{o}] \leftarrow [\boldsymbol{t}] \cdot [\boldsymbol{u}] + [\boldsymbol{o}] \cdot (1 - [\boldsymbol{u}])$
   (d) If round $r \neq n$: $[\boldsymbol{h}] \leftarrow (1 - \mathsf{NotEqualZero}_n([\boldsymbol{o}]))$
   (e) $[\boldsymbol{h}] \leftarrow [\boldsymbol{h}] + \mathsf{rot}([\boldsymbol{h}], n, \mathsf{right})$

[a] This requires steps similar to the Pack Vector sub-routine in Fig. 4.2. Hence, we skip the details here.

Fig. 6: Protocol for secure top trading cycles with SIMD operations.

	Adjacency Matrix	Cycle computation	Update assignments
Additions	$2n^2 + 2n$	$(2n-2)\lceil \log_2(n) \rceil + n + 3$	$n\lceil \log_2(n) \rceil + 2n + 3$
Rotations	$2n^2 + n(\lceil \log_2(n) \rceil + 2)$	$(6n-4)\lceil \log_2(n) \rceil$	$n\lceil \log_2(n) \rceil + n + 1$
Constant Multiplications	$2n$	$(5n-3)\lceil \log_2(n) \rceil$	$2n + 2$
Multiplications	$2n^2 + n(\lceil \log_2(n) \rceil + 2)$	$n(\lceil \log_2(n) \rceil + 1)$	$n + 2$
Mult. Depth	$\lceil \log_2(n) \rceil + 4$	$3\lceil \log_2(n) \rceil + \lceil \log_2(n+1) \rceil$	$\lceil \log_2(n+1) \rceil + 3$

Fig. 7: Complexity measures of $\Pi_{\mathsf{TTC\text{-}SIMD}}$.

A Detailed Related Works

Here, we elaborate on some closely related problems and their privacy preserving approaches.

Matching Algorithms. In the stable marriage problem of Gale and Shapley [GS62] agents are divided in two groups. Each agent in one group has a preference list over all agents in the other group and eventually all agents would like to be matched with an agent from the other group. The original work of Gale and Shapley presents a non-private algorithm for solving the stable marriage problem and subsequently several works have presented privacy-preserving variants of this algorithm [Gol06, FGM07, KS14, ZWR+16, DEs16, MPA+23]. On an intuitive level, the stable marriage problem is harder than the problem we consider, as evidenced by Roth [Rot82a], who showed that any mechanism that solves the stable marriage problem incentivizes at least some of the agents to be dishonest about their true preference list. On a technical level, the ideas for computing stable marriages privately do not appear to be useful for solving the problem considered in this work.

The housing allocation problem of Hylland and Zeckhauser [HZ79] is identical to the one-sided matching markets we consider, with the only difference being that initially nobody owns any of the goods. Abdulkadiroğlu and Sönmez [AS98] proved that randomly assigning goods to agents and then resolving the corresponding one-sided matching market has many appealing game-theoretic properties. Using the above insight, the protocols in our work can easily be adapted to solve the housing allocation problem as well.

In the kidney exchange problem of Roth, Sönmez, and Ünver [RSÜ04], we have n agent-donor pairs. Each agent is in need and each donor provides one kidney, but the individual agent-donor pairs may be incompatible with each other. The goal of a kidney exchange is to trade kidneys in a way that agents receive kidneys they are compatible with. Privacy-preserving kidney exchange protocols have recently been proposed in several works [BHK+22, BHP+22, BMW22, BMW23]. From a technical perspective, these protocols are restricted and tailored to finding very small cycles of trades, usually of length two or three, whereas one-sided matching markets will require finding arbitrarily large cycles in graphs efficiently. For this reason, it seems that the ideas from current privacy-preserving kidney exchange protocols do not translate to our setting.

B Security Model

The real-world experiment is defined in terms of an external distinguisher called the environment $\mathcal{Z}$, an adversary $\mathcal{A}$, parties $\mathcal{P} := \{P_1, \ldots, P_n\}$, and the hybrid functionality $\mathcal{F}'$. $\mathcal{Z}$ writes inputs to all the parties, reads outputs of all the parties, and interacts with $\mathcal{A}$ throughout the experiment execution. When initiated, $\mathcal{A}$ corrupts a subset of parties $\mathcal{P}^* \subseteq \mathcal{P}$ and from then on gets read-only access to their internal state (i.e., it cannot make the corrupt party behave arbitrarily).

$\mathcal{A}$ additionally gets to change the corrupt party's inputs as: $\mathcal{Z}$ asks $\mathcal{A}$ for corrupt party's inputs, and on receiving them writes them on the party's input tape. Parties additionally have access to a local functionality $\mathcal{F}'$. We denote $\mathsf{Real}_{\Pi,\mathcal{F}',\mathcal{A},\mathcal{Z}}$ as the output of $\mathcal{Z}$ when running the real-world experiment.

The ideal-world experiment is defined in terms of $\mathcal{Z}$, an ideal-world adversary called the simulator $\mathcal{S}$, a functionality $\mathcal{F}$ and a set of dummy parties $\tilde{\mathcal{P}}$. Here too, $\mathcal{Z}$ interacts with the (ideal-world) adversary $\mathcal{S}$ throughout the experiment execution. On initiation $\mathcal{S}$ first activates an instance of $\mathcal{A}$ internally and relays communication between $\mathcal{Z}$ and $\mathcal{A}$. As in the real-world experiment, $\mathcal{A}$ might corrupt a subset of parties $\tilde{\mathcal{P}}^*$. Since $\mathcal{S}$ intercepts and relays all communication between $\mathcal{Z}$ and $\mathcal{A}$, when $\mathcal{Z}$ decides the inputs for the corrupt parties via $\mathcal{A}$, $\mathcal{S}$ already knows them. On receiving inputs, each honest party (and $\mathcal{S}$) sends them to $\mathcal{F}$ to receive the output. $\mathcal{S}$ has an additional job of intercepting any call to $\mathcal{F}'$ made by a corrupt party $\tilde{\mathcal{P}}^*$ and simulating $\mathcal{F}'$'s response. We denote $\mathsf{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ as the output of $\mathcal{Z}$ when running the ideal-world experiment.

Definition 1. *Let $\mathcal{F}, \mathcal{F}'$ be an n party functionality and Π be an n party protocol. We say that Π securely realizes $\mathcal{F}$ in the $\mathcal{F}'$-hybrid model and in the presence of passive and static corruptions if for all semi-honest adversaries $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$, such that for all PPT $\mathcal{Z}$, the following holds:*

$$\{\mathsf{Ideal}_{\{\mathcal{F},\mathcal{S},\mathcal{Z}\}}(\boldsymbol{x},\lambda,z)\}_{\boldsymbol{x},\lambda,z} \overset{c}{\approx} \{\mathsf{Real}_{\{\Pi,\mathcal{F}',\mathcal{A},\mathcal{Z}\}}(\boldsymbol{x},\lambda,z)\}_{\boldsymbol{x},\lambda,z}$$

where Ideal and Real are the ideal and real-world experiment defined above, $\boldsymbol{x} := \{x_1, \ldots, x_n\}$ denotes the party's inputs, λ is the security parameter, and z is the auxiliary input.

C Leveled Homomorphic Encryption

A homomorphic encryption scheme [Gen09] is a set of algorithms (KeyGen, $\mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval}$) for plaintext message space $\mathcal{X}$ and ciphertext space $\mathcal{Y}$, where,

1. $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda, \ell)$: Takes the security parameter, and the plaintext vector size ℓ as the input and outputs a $(\mathsf{pk}, \mathsf{sk})$ pair.
2. $c \leftarrow \mathsf{Enc}(\mathsf{pk}, \boldsymbol{m})$: Takes a vector of messages $\boldsymbol{m} \in \mathcal{X}^\ell$ and outputs its encryption $c \in \mathcal{Y}$.
3. $c' \leftarrow \mathsf{Eval}(\mathsf{pk}, f, c_1, ..., c_n)$: Takes as input a set of ciphertexts $c_1, \ldots, c_n$ and an operation $f \in \{\textsc{Add}, \textsc{Mult}, \textsc{Rot}_{i,\mathsf{dir}}\}$, and outputs a ciphertext c'. Here, $\textsc{Add}$, $\textsc{Mult}$ and $\textsc{Rot}_{i,\mathsf{dir}}$ denote point-wise addition, point-wise multiplication, and rotation by i slots in the direction dir, resp.
4. $m' \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$: Takes as input a ciphertext and outputs its decryption m'.

A homomorphic encryption scheme is said to be d-leveled (or somewhat homomorphic) if the KeyGen algorithm takes an additional parameter d as input, where d is the maximum allowed multiplicative depth of the arithmetic circuit that can be evaluated over the ciphertexts.

We require that a leveled-HE (or SHE) scheme satisfies the standard notions of correctness, IND-CPA security, and circuit privacy.

D Ideal Functionality: Arithmetic Black Box

The functionality F_{ABB} (Fig. 8) receives commands from two types of computing devices: from clients it receives an input vector of fixed length ℓ, and then it allows servers $S_1, \ldots, S_m$ to securely perform additions, multiplications, and rotations over these inputs. Since the input is a vector of length ℓ, additions and multiplications are then performed *element-wise*. Thus, ℓ parallel additions or multiplications can be computed at the cost of a single call to F_{ABB}. Servers can securely *cycle* vector elements by any number of slots and direction; this operation is called *rotation*, and permits the computation over values stored at different vector indices.

F_{ABB}

F_{ABB} interacts with client $\mathcal{C} = \{C_1, \ldots, C_n\}$ and servers $\mathcal{S} = \{S_1, ..., S_m\}$ and performs arithmetic operations over field domain $\mathbb{Z}_p$. The functionality is parameterized by the length of input vectors ℓ.

Input: On receiving $(\textsc{Input}, \mathsf{sid}, \mathsf{id}, X \in \mathbb{Z}_p^\ell)$ from a client P_i, and $(\textsc{Input}, \mathsf{sid}, \mathsf{id})$ from all other clients, check if $|X| = \mathsf{len}$. If yes, then store (id, X).

Addition: On receiving $(\textsc{Add}, \mathsf{sid}, \mathsf{id}_0, \mathsf{id}_1, \mathsf{id}_2)$ from all servers in $\mathcal{S}$, retrieve (id_1, X_1) and (id_2, X_2). Store $(\mathsf{id}_0, X_1 + X_2)$[a].

Multiply: On receiving $(\textsc{Mult}, \mathsf{sid}, \mathsf{id}_0, \mathsf{id}_1, \mathsf{id}_2)$ from all servers in $\mathcal{S}$, retrieve (id_1, X_1) and (id_2, X_2). Store $(\mathsf{id}_0, X_1 \cdot X_2)$.

Rotate: On receiving $(\textsc{Rot}, \mathsf{sid}, \mathsf{id}_0, \mathsf{id}_1, s, \mathsf{dir})$ from all servers in $\mathcal{S}$, retrieve (id_1, X). Parse $X := (x_1, \ldots)$. If $\mathsf{dir} := \mathsf{left}$, then rotate to the left by s indices, i.e., $X' := (x_{s+1}, \ldots, x_s)$. Else, if $\mathsf{dir} := \mathsf{right}$, rotate to the right by s. Store (id_0, X').

Open-to-C: On receiving $(\textsc{Open}, \mathsf{sid}, \mathsf{id}, C)$ from all clients in $\mathcal{C}$ and servers in $\mathcal{S}$, retrieve (id, X) and return X to the client C and $(\textsc{Opened})$ to all other clients.

[a] Both Add and Mult operations are performed element-wise.

Fig. 8: Arithmetic black box functionality

Realizing F_{ABB} In this work we consider the client-server setting with n clients and m servers. Specifically, we consider that one server (say, S_1) performs the homomorphic operations over ciphertexts, while the other servers participate in decryption. In this setting, it is easy to see that an leveled-HE encryption scheme UC-realizes F_{ABB} in the $F_{\mathsf{KeyGenDec}}$-hybrid world. The $F_{\mathsf{KeyGenDec}}$ functionality (Fig. 9) allows servers and clients to obtain pk for the leveled-HE scheme where the associated sk is stored inside the functionality. Given pk, clients can encrypt secret inputs and send them to S_1, who then evaluates homomorphic operations over ciphertexts locally with Eval algorithm of the leveled-HE scheme.

Importantly, leveled FHE schemes permit a bounded number of homomorphic operations only; concretely, the noise in ring-LWE based schemes grows with each homomorphic operation, in particular with multiplications. The growth in noise eventually prevents the correct decryption of ciphertext. When evaluating circuits with higher multiplicative depth than what can be tolerated by the leveled-HE scheme, ciphertexts must be refreshed. To refresh, servers initialize a random mask and send its encryption to S_1, who applies the mask to the ciphertext. Servers jointly open it to S_1 by calling $F_{\mathsf{KeyGenDec}}$ (via DECRYPT-TO-S command). S_1 can then re-encrypt and remove the mask via one local addition operation, and continue with the evaluation on a fresh ciphertext. Finally, when the clients want to learn the outcome, they query $F_{\mathsf{KeyGenDec}}$ (via DECRYPT-TO-C command) which will decrypt the ciphertext and output the result to the respective client.

We restate the formal security theorem below (Thm 4). The proof for is similar to that in [DPSZ12], but for completeness present the main ideas below.

Theorem 4. *Let* $\mathsf{LHE} := (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ *be a* leveled-HE *scheme that is correct,* IND-CPA *secure, and is circuit private. Then* F_{ABB} *can be UC-realized by a protocol with* LFHE *in the* $F_{\mathsf{KeyGenDec}}$*-hybrid against any static, passive adversary corrupting up to* $m-1$ *servers and* $n-1$ *clients.*

Proof. We assume private communication channels between all the servers and clients. We also assume that the environment issues consistent commands (as expected by the functionality) to all the clients and servers. Otherwise, it would be possible to account for inconsistent commands by adding additional checks as a part of the honest protocol execution, and in F_{ABB} description.

Honest clients and servers behave as follows. On initialization, clients and servers call $F_{\mathsf{KeyGenDec}}$ and wait to receive pk. On receiving $(\textsc{Input}, \mathsf{sid}, \mathsf{id}, X)$, a client computes ciphertext $c \leftarrow \mathsf{Enc}(X, \mathsf{pk})$, stores (id, c), and sends (id, c) to server $\mathcal{S}_1$ and (id) to the rest of the servers. From here on, for all operational commands (ADD, MULT, and ROT) received by the serves, $\mathcal{S}_1$ executes them locally after checking if the corresponding ciphertexts and ids are stored. If a ciphertext $(\mathsf{id}, c_)$ needs to be refreshed, before evaluating on it, servers send an encryption of a random mask to S_1, who uses them to mask the ciphertext and open it by calling $F_{\mathsf{KeyGenDec}}$ (via DECRYPT-TO-S command). S_1 can then freshly encrypt and remove the mask with one local addition operation, store the new ciphertext under the same (id) as before, and continue with the evaluation.

The simulator's algorithm (Sim) is straight forward. Recall that in the semi-honest UC setting, all inputs for the corrupt parties are forwarded via Sim. This means that all INPUT commands for the corrupt clients are known to Sim, and so it can call F_{ABB} on the corrupt clients' behalf. Moreover, since we are in $F_{\mathsf{KeyGenDec}}$-hybrid, Sim will intercept corrupt client and servers' calls to this functionality and simulate its responses. Sim behaves as follows.

– **Initialization:** On initialization the servers and the clients call $F_{\mathsf{KeyGenDec}}$ on INIT command. Sim receives all INIT calls, sample pk, sk and outputs pk to all. Sim invokes an instance of $\mathcal{A}$ internally and relays all communication between $\mathcal{Z}$ and $\mathcal{A}$. Let $\tilde{\mathcal{C}}^*$ and $\tilde{\mathcal{S}}^*$ be the clients and servers corrupted by $\mathcal{A}$.

– **Simulating input command:** When $\mathcal{Z}$ sends $((Input), \cdot, X_i)$ intended for a corrupt client $\tilde{C}_i^*$, pass it to $\mathcal{A}$. On receiving X_i' in response from $\mathcal{A}$, relay it to $\mathcal{Z}$. Note that the client $\tilde{C}_i^*$ will now be initialized with input X_i'. Forward $((Input), \cdot, X_i)$ to F_{ABB}, and send an encryption to the corrupt server. When $\mathcal{Z}$ sends $((Input), \cdot)$ to the corrupt party (when the input is intended for an honest party), encrypt 0^ℓ and send it to the corrupt server on behalf of the honest client.
– **Simulating add,mult,rot command:** On receiving an operation command form $\mathcal{Z}$ on behalf of a corrupt server, forward it to F_{ABB}.
– **Simulating Open-to-C command:** On receiving $(\textsc{Open}, \cdot, \cdot, C)$ command where C is corrupt, forward $(\textsc{Open}, \cdot, \cdot, C)$ to F_{ABB} on the corrupt client and server's behalf. Receive the opening X in response. Receive query to $F_{\mathsf{KeyGenDec}}$ on (DECRYPT-TO-C) from the corrupt entities and forward X to the corrupt client C.
– **Simulating refreshes for a corrupt S_1 server:** Sim encrypts a random mask vector on honest server's behalf and send them to S_1. Then, intercept the DECRYPT-TO-S command made by the corrupt servers, and open a random vector to S_1.

Indisintguishability of the view generated by Sim follows from correctness and IND-CPA properties: correctness ensures that the decryption of a ciphertext obtained by applying the Eval algorithm (as in the real protocol execution) is the same as the output of the evaluation over plaintext (as in the ideal protocol); and IND-CPA ensures that ciphertexts for $\mathbf{0}$ vectors generated by Sim (as in the ideal protocol) look indistinguishable from the real ciphertexts generated by the honest parties (as in the real protocol). We would like to remark that since we are in the UC-setting we must assume some setup, and we chose to rely on $F_{\mathsf{KeyGenDec}}$ functionality (see [DPSZ12] for a similar treatment).

□

$F_{\mathsf{KeyGenDec}}$

$F_{\mathsf{KeyGenDec}}$ interacts with clients $\mathcal{C} = \{C_1, ..., C_n\}$ and servers $\mathcal{S} = \{S_1, \ldots, S_m\}$ and is parameterized by the underlying leveled-HE encryption scheme $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$.

Init: On $(\textsc{Init}, \mathsf{sid})$ from all clients and servers, compute $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$, and send $(\mathsf{sid}, \mathsf{pk})$ to all. Ignore all future commands.

Decrypt-to-C: On $(\textsc{Decrypt-to-C}, \mathsf{sid}, c, C_i)$ from server $\mathcal{S}_1$, and $(\textsc{Decrypt-to-C}, \mathsf{sid}, C_i)$ from all clients and all other servers, compute $m \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$ and output m to C_i and $\textsc{Opened}$ to all other clients.

Decrypt-to-S: On $(\textsc{Decrypt-to-S}, \mathsf{sid}, c)$ from server $\mathcal{S}_1$, and $(\textsc{Decrypt-to-S}, \mathsf{sid})$ from all clients and all other servers, compute $m \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$ and output m to S_1 and $\textsc{Opened}$ to all other servers.

Fig. 9: Distributed key generation and decryption functionality

E Proof of Lemma 1

Lemma 3. *Let M be the adjacency matrix of a functional graph with n nodes, let $\mathbf{1}$ be a column vector where all entries are 1 and let $\boldsymbol{u} = M^n\mathbf{1}$. Then $\boldsymbol{u}_i$ is non-zero if and only if node i is on a cycle. Moreover, $\boldsymbol{u}_i$ is a non-negative integer and $\boldsymbol{u}_i \leq n$.*

Proof. Observe first that $\boldsymbol{u}_i$ is the number of paths of length n that end in node i. It is therefore of course a non-negative integer and since we have a functional graph, for every node j, there can be at most one path of length n from j to i, whence $\boldsymbol{u}_i \leq n$.

As for the cycle property, note that if node i is on a cycle, we can construct a path of length n ending in node i, by simply going n steps "backwards" in the cycle. So this means $\boldsymbol{u}_i > 0$ in this case.

Finally, assume node i is not on a cycle and consider a node j where there is a edge from j to i. Note that j cannot be on a cycle, as this would imply that the out-degree of j would be greater than 1. We can repeat the same argument for any node "pointing to" j. Continuing in this way, we conclude for every node ℓ where there is a path from ℓ to i, that ℓ is not on a cycle. This means that no path that ends in node i can contain the same node more than once, and hence any such path has length at most $n - 1$, so that $\boldsymbol{u}_i = 0$ in this case. □

F Correctness and Security of Π_{TTC}

Lemma 4. *(Correctness) Π_{TTC} (Figure 2) implements the top trading cycle algorithm (specified in F_{TTC}).*

Proof. Correctness of step (1) in Π_{TTC} (deriving the adjacency matrix from client preferences) and step (2) (computing cycles from adjacency matrix) follow from

Sec 3 and Lem 1 respectively. It remains to argue correctness of step (3) of Π_{TTC} over n rounds, which updates the availability and assigned output for clients. We must show that once a client is assigned to a cycle, its output is set and that its good is no longer available for the remainder of the protocol.

For convenience, we reproduce protocol steps 3.a.ii. and 3.b.i. which update output and availability of each client following cycle computation in each round.

- 3.a.ii. $\boldsymbol{o}_i \leftarrow \boldsymbol{t}_i \cdot \boldsymbol{u}_i + \boldsymbol{o}_i \cdot (1 - \boldsymbol{u}_i)$
- 3.b.i. $\boldsymbol{h}_i \leftarrow (1 - \mathsf{NotEqualZero}(\boldsymbol{o}_i))$

Here, $\boldsymbol{u}_i$ is a bit which is 1 if client i is on a cycle and 0 otherwise, and $\boldsymbol{t}_i$ is the index of client i's most preferred available good at the onset of the round. Outputs ($\boldsymbol{o}$) are initialized to 0^n and availability of clients ($\boldsymbol{h}$) to 1^n.

Cycle finding round 1. Following the first cycle computation, clients are either on a cycle $\boldsymbol{u}_i = 1$ or not $\boldsymbol{u}_i = 0$ (Step 2). We distinguish these two cases:

- $\boldsymbol{u}_i = 1$: Client i has its output $\boldsymbol{o}_i$ updated to its top preference $\boldsymbol{t}_i$ (step 3.a.ii). Further, its availability $\boldsymbol{h}_i$ for the subsequent round is set to "0" (Step 3.b.i). Therefore, client i's good is never considered in subsequent updates of the adjacency matrix; in graph representation, the client will have no inbound edges.
- $\boldsymbol{u}_i = 0$: Client i's output $\boldsymbol{o}_i = 0$ remains as initialized (step 3.a.ii) and its availability $\boldsymbol{h}_i$ remains "1" (step 3.b.i).

Cycle finding round $1 < r \leq n$. In each subsequent round, we distinguish between two cases for each client i dependent on the value assigned to its output variable $\boldsymbol{o}_i$ at the *onset* of round r.

- $\boldsymbol{o}_i \neq 0$ at onset of round r. In this case, the availability of client i must be $\boldsymbol{h}_i = 0$ at the onset of the round, otherwise $\boldsymbol{o}_i = 0$ must hold according to step 3.b.i. (contradiction). Naturally, client i cannot be assigned in the cycle finding round without availability; it remains in case "$\boldsymbol{o}_i \neq 0$" for all subsequent rounds as its availability $\boldsymbol{h}_i$ will always be assigned 0 for $\boldsymbol{o}_i \neq 0$ in step 3.b.i.
- $\boldsymbol{o}_i = 0$ at onset of round r. In this case, the availability of client i must be $\boldsymbol{h}_i = 1$ at the onset of the round. Otherwise $\boldsymbol{o}_i$ must be non-zero according to step 3.b.i. (contradiction). Following the cycle finding steps (1) and (2) in round r in the case that client i's good was available at the onset of the round, we distinguish the following subcases following cycle computation in round r.
  - $\boldsymbol{u}_i = 1$. If client i is assigned to a cycle in round r, its output will subsequently be updated $\boldsymbol{o}_i \neq 0$, and its availability is set to $\boldsymbol{h}_i = 0$. The client cannot be assigned to a cycle and its non-zero output $\boldsymbol{o}_i$ is no longer updated. It remains in case $\boldsymbol{o}_i \neq 0$ for subsequent rounds.
  - $\boldsymbol{u}_i = 0$. If client i is not assigned to a cycle in a round, its output remains as initialized ($\boldsymbol{o}_i = 0$) and it remains available for the next round ($\boldsymbol{h}_i = 1$).

Thus, once a client i has been assigned a cycle for the first time, its output variable remains set until the end of the protocol and its good is removed from the pool of available parties in all subsequent cycle computations. □

Theorem 5. *Π_{TTC} (Figure 2) UC-realizes F_{TTC} (Figure 1) in the F_{ABB}-hybrid model against a passive adversary corrupting up to $m-1$ servers and $n-1$ clients.*

Proof. Note that the clients or the servers never interact with each other directly, and F_{ABB} only produces a response for an OPEN-TO-P commands. Hence the view of $\mathcal{Z}$ in the real-world experiment consists of just the inputs and outputs of all the parties. Thus, in the ideal-world, the simulator Sim's job is simply to simulate responses from F_{ABB} for the corrupt servers, and open the final output correctly to a corrupt clients. The simulator Sim works as follows: On initialization, run $\mathcal{A}$ internally and relay all communication between $\mathcal{Z}$ and $\mathcal{A}$. Let $\tilde{\mathcal{C}}^*$ and $\tilde{\mathcal{S}}^*$ be the clients and servers corrupted by $\mathcal{A}$, resp. From $\mathcal{Z}$, on receiving $((Input), \boldsymbol{x}_i)$ for a corrupt client $\tilde{C}_i^*$, pass it on to $\mathcal{A}$. On receiving $\boldsymbol{x}_i'$ from $\mathcal{A}$, relay it back to $\mathcal{Z}$. Note that the client $\tilde{C}_i^*$ will now be initialized with input $\boldsymbol{x}_i'$. Sim sends $\boldsymbol{x}_i'$ to F_{TTC} on behalf of $\tilde{C}_i^*$ to receive output_i. To simulate $\mathcal{A}$'s view, receive calls to F_{ABB} for all commands made by a corrupt server and respond with nothing. Finally, receive call to OPEN from a corrupt client $\tilde{C}_i^*$ and respond with output_i. The view in the real-world experiment consists of just the input and the final output (since all interaction is via F_{ABB}). Hence, indinstinguishability of the view generated by Sim is implied by correctness (argued in Lemma 2). □

Theorem 6. *Protocol Π_{TTC} evaluated on n client preference lists privately input to hybrid functionality F_{ABB} incurs a maximum multiplicative depth of $O(n\log(n))$ on values output from F_{ABB}.*

Proof. We illustrate the multiplicative depth complexity of Π_{TTC} for a single round.

- **Adjacency matrix ($2 + \lceil\log_2(n)\rceil$).** Steps (1.a.i) and (1.a.iii) represent matrix-vector mulplications, each incrementing the maximum multiplicative depth of values stored in F_{ABB} by 1. PreserveLeadOne incurs additional multiplicative depth of $\lceil\log_2(n)\rceil$.
- **Cycle computation($\lceil\log_2(n)\rceil+\lfloor\log_2(n)\rfloor+\lceil\log_2(\lfloor\log_2(n)\rfloor)\rceil$).** Exponentiate requires $\lfloor\log_2(n)\rfloor + \lceil\log_2(\lfloor\log_2(n)\rfloor)\rceil$ and NotEqualZero incur multiplicative depth of $\lceil\log(n)\rceil$.
- **Assignment & availability ($2 + \lceil\log_2(n)\rceil$).** Step 3.a.i and 3.a.ii each increment multiplicative depth by 1. Step 3.b incurs $\lceil\log(n)\rceil$ via NoEqualZero.

Thus, over n rounds, we incur a maximum of multiplicative depth of $O(n\log(n))$ on values in F_{ABB}. □

G Detailed TTC-SIMD protocol

In this section, using the SIMD-style operations exposed by F_{ABB} and realized by the leveled-HE frameworks (Appendix D), we illustrate SIMD algorithms over vectors for all sub-protocols in in $\Pi_{\mathsf{TTC\text{-}SIMD}}$ (Figure 6).

The overall template of protocol $\Pi_{\mathsf{TTC\text{-}SIMD}}$ (Fig. 6) follows that of Π_{TTC} (Fig. 2), but extends it by integrating various SIMD algorithms described in Section. 4.2 to achieve improved concrete complexity over Π_{TTC} in Table 3.

Correctness of $\Pi_{\mathsf{TTC-SIMD}}$ is implied by that of Π_{TTC}; for example, we note that parties in $\Pi_{\mathsf{TTC-SIMD}}$ only make calls to F_{ABB} and have no other communication directly with each other. Thus, the security arguments from Π_{TTC} apply directly to $\Pi_{\mathsf{TTC-SIMD}}$. Complexity measures of $\Pi_{\mathsf{TTC\text{-}SIMD}}$ are given in Fig. 7.

A concrete challenge addressed by $\Pi_{\mathsf{TTC\text{-}SIMD}}$ is to integrate various SIMD algorithms introduced in Section **??** which operate over different vector domains, whilst F_{ABB} must be initialized for a fixed vector domain. For example,

- NotEqualZero (Eq. 4) is evaluated element-wise, and can thus be applied to vectors of any length.
- PrefixAdd/Mult (Fig. 11) computes prefix arithmetic over n leading values in a vector of length $n' \geq 2^{k+1} - 1$, where $k = \lceil \log_2(n) \rceil$; the padding of 0s or 1s following the n leading "active" vector elements can thus be extended for any compliant n'. The same follows for InnerProduct, which is an element-wise multiplication followed by PrefixAdd. Note that these operations result in a vector with elements in indices $i > n$ filled with intermediary values, which must be accounted for in the full protocol.
- PreserveLeadOne (Eq. 6) also involves evaluating PrefixMult over n elements, but requires a vector dimesion of $n' \geq 2^{k+1}$ as a subsequent right rotation requires that there is one trailing 1-element in the vector that is cycled into the first vector index.
- Matrix-Vector Mult. (Eq. G.3) are over vectors of length n, where matrix diagonals are encoded as separate vectors.
- Matrix-Matrix Mult. (Eq. 9) are over vectors of length n^2, where all matrix elements are encoded in a single vector.

G.1 Protocol overview

We now provide a step-by-step description of $\Pi_{\mathsf{TTC\text{-}SIMD}}$ in Fig 6, which forwards calls to F_{ABB} parameterized over integer vectors of dimension n^2. In the next section, we will explain the details of each sub-protocol used $\Pi_{\mathsf{TTC\text{-}SIMD}}$.

In *update adjacency matrix*, we first multiply the availability vector with the permutation matrix of each client to obtain the availability of goods in order of its preference; note that the matrix-vector multiplication in step 1.a.i. of Fig 6 is evaluated over the first n vector elements. Thus, the availability matrix $\boldsymbol{h}$ is replicated once to the right (during initialization and its update in step 3.e.), to emulate the cycling of elements over lower vector dimension n. In step 1.a.ii. we isolate or mask the first n vector elements, since active elements in $\boldsymbol{h}$ are replicated and now populate elements in indices outside of the leading n of $\boldsymbol{a}^{(i)}$. Furthermore, step 1.a.ii. pads the vector with trailing 1s, necessary for the correct evaluation of PreserveLeadOne in 1.a.ii. The resulting vector $\boldsymbol{b}^i$ with a single 1 must then be masked (1.a.iv) and replicated (1.a.v.) to emulate correct element cycling of n elements in matrix-vector multiplication of step 1.a.vi.

In *compute cycles*, each matrix is encoded as a single vector in F_{ABB}. Thus, it is necessary to re-encode the rows of the adjacency matrix from step (1) into a single flat-encoded vector in step 1.b. Matrix squaring in 2.b. is then evaluated over vectors of dimension n^2. We note that $\lceil \log_2(n) \rceil$ squarings are sufficient for cycle-finding; the proof argument of Lemma. 1 trivially holds. However, subsequent multiplication of the squared adjacency matrix with the $\mathbf{1}$ vector will require re-encoding of the matrix in diagonal form (step 2.c).

In *update assignments and availability*, we compute the index of the most preferred available good of each party in step 3.a.; $\mathsf{InnerProd}_{\mathsf{R}}$ over $\boldsymbol{w}^{(i)}$ and $[(1, 2, ..., n)]$ brings the desired index to the first element of the vector. We then encode all preferred indices in a single vector in 3.b., permitting the update of assignments and availability in purely element-wise fashion in 3.c-3.e.

G.2 SIMD on vectors of different lengths.

We note that F_{ABB} must be parameterized by a single vector length ℓ; all subsequent arithmetic operation and rotation calls must then operate over vectors of length ℓ. Subsequently described SIMD-style algorithms, however, are over vector lengths of n, $2n$ and n^2; different vectors must be carefully padded or replicated to ensure correctness over ℓ, which is set to n^2 in F_{ABB} initialized by $\Pi_{\mathsf{TTC\text{-}SIMD}}$ (Figure 6). When we write that a client calls F_{ABB} with command $(\textsc{Input}, \mathsf{sid}, \mathsf{id}, \boldsymbol{v})$ and $|\boldsymbol{v}| \leq \ell$ then it is implicitly assumed that, before calling F_{ABB}, the client pads $\boldsymbol{v}$ with appropriate number of zeros so that $|\boldsymbol{v}, \mathbf{0}| = \ell$. For simplicity, following SIMD-style algorithms will be described in their "native" vector dimension; we refer to Appendix G for details on how they are realized in $\mathbb{Z}^{n^2}$.

G.3 Sub-Protocols of $\Pi_{\mathsf{TCC\text{-}SIMD}}$

Mask./Pad./Repl. of vector elements. $\Pi_{\mathsf{TCC\text{-}SIMD}}$ initializes F_{ABB} to securely perform SIMD operations over vector lengths of n^2; this requires careful masking, padding and replication/copying of vector elements stored in F_{ABB} to ensure correctness of PrefixMult, InnerProd, PreserveLeadOne, Matrix-Vect over $\mathbb{Z}^{n^2}$. In general, all inputs are padded with 0's to fill a vector of length n^2.

Pack vectors. Protocol $\Pi_{\mathsf{TCC\text{-}SIMD}}$ encodes matrices row-wise (adjacency matrix update), as diagonals (matrix-vector multiplication), and in flattened form (matrix-matrix multiplication). To re-encode matrices from one form to another whilst minimizing (expensive) multiplication and rotation calls to F_{ABB}, we simply mask matrix elements, and then open the masked values; subsequently, the masked matrix elements are re-arranged and input again to F_{ABB} in the desired encoding form, where they are unmasked inside F_{ABB}. We note that random masks can be generated by aggregating randomness contributions from servers prior to the client input phase, thereby not affecting the overall protocol runtime for clients. When realizing F_{ABB}, the opening of masked values will naturally

incur communication; in practice, this communication can be chosen to coincide with ciphertext refreshing (Section 5), and thus does not incur any additional communication complexity.

For example, to re-encode a matrix stored as separate vectors in F_{ABB}, where each matrix row occupies leading n indices of each n^2 vector, to a matrix with all rows in a single vector of length n^2, the protocol evaluates *Pack Vectors* as shown in Fig. 10. To re-encode matrix rows $[\boldsymbol{w}_1, 0, ..., 0], \ldots, [\boldsymbol{w}_n, 0, ..., 0]$ as $[\boldsymbol{w}] := [\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n]$, the naive approach would be to rotate each vector by the appropriate positions ($i \cdot n$ positions for the i-th vector) and then add them up. This results in $O(n)$ rotations. Instead, the parties initialize masks $([\boldsymbol{r}_1, 0, ..., 0], \ldots, [\boldsymbol{r}_n, 0, ..., 0])$, locally mask each $[\boldsymbol{w}_i]$ (with $\boldsymbol{r}_i$) and open the masked vector to one of the parties (say P_1). Then P_1 can locally rearrange all masked vectors in the correct order and reinitialize it as one vector. To use this further, the parties must remove the masks. This can be done easily if the parties, while initializing $\boldsymbol{r}_i$ (used to mask $\boldsymbol{w}_i$), also initialize a rotated version of it so that it aligns with wherever $\boldsymbol{w}_i$ lies in $\boldsymbol{w}$ after packing.

Pack Vectors: matrix rows

This sub-routine packs the first n slots of vectors $([\boldsymbol{w}^{(1)}], \ldots, [\boldsymbol{w}^{(n)}])$ as one vector $[\boldsymbol{w}]$.

1. For each $i \in [n]$,
   (a) Each party P_j samples $\boldsymbol{r}_j^{(i)} \leftarrow \mathbb{F}_p^n$ at random and computes $\boldsymbol{R}_j^{(i)} := \{0^{j \cdot n}, \boldsymbol{r}_j^{(i)}, 0, \ldots 0\}$. P_j initializes $\boldsymbol{r}_j^{(i)}, \boldsymbol{R}_j^{(i)}$ with F_{ABB}.
   (b) $[\boldsymbol{r}^{(i)}] \leftarrow \sum_{i=1}^{n} [\boldsymbol{r}_j^{(i)}]$ and $[\boldsymbol{R}^{(i)}] \leftarrow \sum_{i=1}^{n} [\boldsymbol{R}_j^{(i)}]$.
   (c) $[\boldsymbol{m}^{(i)}] \leftarrow [\boldsymbol{w}^{(i)}] + [\boldsymbol{r}^{(i)}]$.
   (d) All parties call F_{ABB} with input $(\textsc{Open}, \cdot, P_1)$ to allow P_1 to learn $\boldsymbol{m}^{(i)}$.
2. Party P_1 computes $\boldsymbol{m} := (\boldsymbol{m}^{(1)}, \ldots, \boldsymbol{m}^{(n)})$ and initializes it with F_{ABB}.
3. $[\boldsymbol{R}] \leftarrow \sum_{i=1}^{n} [\boldsymbol{R}^{(i)}]$
4. $[\boldsymbol{w}] \leftarrow [\boldsymbol{m}] - [\boldsymbol{R}]$

Fig. 10: Sub-routine to pack n matrix rows in one vector.

We note that the exponentiated matrix in the cycle computation step must also be re-encoded in form of its diagonals (Eq. 7); here, we follow the same approach as in Fig. 10, but omit an formal description for brevity.

NotEqualZero. We implement a mapping of positive integers in range $[0, n]$ to 1 and note that all input values in our application satisfy this range (Lemma 1). NotEqualZero is a pure SIMD algorithm, that computes element-wise only; it does not call rotations in F_{ABB}. Each vector index represents a separate parallel execution. To compute NotEqualZero for all elements in input vector $[\boldsymbol{v}]$, we

evaluate;

$$\mathsf{NotEqualZero}_n([\boldsymbol{v}]) = [\mathbf{1}] - [\boldsymbol{n!^{-1}}] \prod_{i \in [n]} ([\boldsymbol{i}] - [\boldsymbol{v}]) \tag{4}$$

Here, let $[\boldsymbol{i}]$ and $[\boldsymbol{n!^{-1}}]$ denote $[i, ..., i]$ and $[n!^{-1}, ..., n!^{-1}]$ with the same dimension as the input vector. Note that these two vectors can be precomputed locally and initialized by a server. For correctness, consider vector element v_j at index j and observe that if $v_j > 0$ in range $[1, n]$, then the product term $\prod_{i \in [1,n]}(i - v_j)$ is 0. Conversely, if v_j is zero, then the product term will equal $n!$. The output of $\mathsf{NotEqualZero}$ in the first case is 1, while in the latter case is 0. By moving the factor -1 out of the product term and considering cases of even or odd n, we rewrite as follows to avoid negation by multiplication;

$$\mathsf{NotEqualZero}_n([\boldsymbol{x}]) = \begin{cases} [\mathbf{1}] + [\boldsymbol{-n!^{-1}}] \prod_{i \in [n]}([\boldsymbol{x}] + [\boldsymbol{-i}]) & n \text{ even} \\ [\mathbf{1}] + [\boldsymbol{n!^{-1}}] \prod_{i \in [n]}([\boldsymbol{x}] + [\boldsymbol{-i}]) & n \text{ odd} \end{cases} \tag{5}$$

$\mathsf{NotEqualZero}$ in SIMD-fashion and input range $[0, n]$ incurs a total multiplicative and additive complexity of $n + 1$ and consumes multiplicative depth of $\lceil \log_2(n + 1) \rceil$.

PrefixAdd/PrefixMult. $\mathsf{PrefixAdd}_{\mathsf{L/R}}$ and $\mathsf{PrefixMult}_{\mathsf{L/R}}$ algorithms allows parties to compute sum/product of all prefixes of a vector with only $\log(n)$ additions/multiplications and rotations. These prefix algorithms are essential to implement $\mathsf{InnerProd}$ and $\mathsf{PreserveLeadOne}$ with concrete efficiency.

We adapt parallel prefix arithmetic from [HSJ86] to the setting of SIMD operations in F_{ABB}. Suppose we want to compute the prefix sum of the vector $[1, 2, ..., 8]$ shown in the upper left of Figure 11; $\mathsf{PrefixAdd}_{\mathsf{L}}$ outputs a vector where each element holds the sum of all input elements with indices that are equal or lower. $\mathsf{PrefixAdd}_{\mathsf{R}}$ outputs a vector where each element holds the sum of all input elements with indices that are equal or higher.

The n-element input vector is padded with 0's to length $n' \geq 2^{k+1} - 1$, where $k = \lceil \log_2(n) \rceil$; subsequent SIMD operations are over this extended domain which accommodates the trailing 0s. Our example vector in Figure 11 is padded to $[1, 2, ..., 8, 0^7]$. To then evaluate $\mathsf{PrefixAdd}_{\mathsf{L}}$, we proceed by levels. For level $i \in [0, \lceil \log_2(n) \rceil - 1]$, we rotate the intermediate vector to the *right* by 2^i slots (see left of Figure 11). The unrotated vector from the preceding level is then added element-wise in SIMD-fashion with a single addition call to F_{ABB}. After $\lceil \log_2(n) \rceil$ levels, we obtain the resultant vector where each element holds the sum of all elements of the input vector with lower slot indices. This requires only $O(\log_2(n))$ total additions, and rotations. $\mathsf{PrefixAdd}_{\mathsf{R}}$ is evaluated in the same manner with rotations to the *left* at each level.

Evaluating $\mathsf{PrefixMult}_{\mathsf{L/R}}$ requires extending the input vector of length n with $2^{\lceil \log_2(n) \rceil + 1} - 1$ trailing 1's. The rest remains the same as in prefix addition, albeit with multiplication operations instead of additions. Here too, the algorithm incurs $\lceil \log_2(n) \rceil$ total multiplications and rotations. In contrast, evaluating prefix multiplication over individually encrypted values would incur $O(n)$ multiplications.

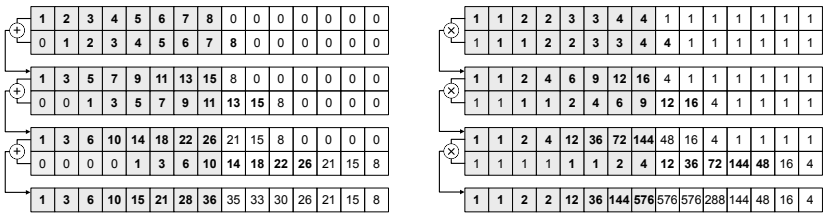

Fig. 11: $\mathsf{PrefixAdd}_{\mathsf{L}}$ and $\mathsf{PrefixMult}_{\mathsf{L}}$. The top row represents the padded input vector - at each level $i \in [0, \log_2(n) - 1]$, the vector is rotated by 2^i steps and added/multiplied with the preceding vector. Each element in the final vector holds the aggregate sum/product of input vector elements with indices to the left.

$\mathsf{PrefixAdd}_{\mathsf{D}\in\{\mathsf{L},\mathsf{R}\}}/\mathsf{PrefixMult}_{\mathsf{D}\in\{\mathsf{L},\mathsf{R}\}}$

Input: For $\mathsf{PrefixAdd}$, input $[\boldsymbol{v}] = [v_1, \ldots, v_{n=2^k}]$ is extended to $[v_1, \ldots, v_n, 0, ..., 0] \in \mathbb{Z}^{n'}$ for $n' \geq 2n - 1$. For $\mathsf{PrefixMult}$, trailing 1s are appended.

Compute Prefix: For $\mathsf{lvl} \in [1, \lceil \log(n) \rceil]$, do:

1. If $\mathsf{D} = \mathsf{L}$, $\mathsf{rotD} \leftarrow \mathsf{R}$. Else, $\mathsf{rotD} = \mathsf{L}$.
2. $[\boldsymbol{u}] \leftarrow \mathsf{rot}([\boldsymbol{v}], 2^{\mathsf{lvl}}, \mathsf{rotD})$
3. $[\boldsymbol{v}] \leftarrow [\boldsymbol{v}] + [\boldsymbol{u}]$ (resp. $[\boldsymbol{v}] \leftarrow [\boldsymbol{v}] \cdot [\boldsymbol{u}]$)

Output: $[\boldsymbol{v}]$

Fig. 12: Protocol for computing Prefix Sums.

InnerProduct. The $\mathsf{InnerProd}$ over n-sized vectors packed in ciphertexts $[\boldsymbol{v}]$ and $[\boldsymbol{w}]$ can be computed with a single multiplication and evaluating the additive prefix over the resulting ciphertext $[\boldsymbol{v}] \cdot [\boldsymbol{w}]$, namely

$$\mathsf{PrefixAdd}_{\mathsf{R}}([\boldsymbol{v}] \cdot [\boldsymbol{w}]) = \left[\sum_{i \in [m]} v_i w_i, \sum_{j \in [m-1]} v_j w_j, ... \right]$$

The inner product scalar is then located in the first slot position, and if necessary can be *extracted* by multiplying with a fresh encryption of $[1, 0, ...]$. The $\mathsf{InnerProd}$ operation over vectors thus incurs a total multiplicative complexity of 1 (or 2 if extraction is required).

PreserveLeadOne. This algorithm takes as input a vector $\boldsymbol{x} \in \{0, 1\}^n$ and preserves the first 1, while zeroing out the rest of the vector. $\mathsf{PreserveLeadOne}$ requires padding with 1s to the right such that $[\boldsymbol{x}, 1, ..., 1]$ is of dimension $n' \geq 2^{\lceil \log_2(n) \rceil + 1}$; one additional trailing 1 is required for correctness due to the required rotation by a single index. The following are therefore SIMD operations over

vectors of length n'.

$$[\boldsymbol{x}, 1, ..., 1] \cdot \mathsf{rot}(\mathsf{PrefixMult}_{\mathsf{L}}([\mathbf{1}] - [\boldsymbol{x}, 1, ..., 1]), 1, \mathsf{right}) \tag{6}$$

Correctness follows from Section 3 on preference computation. PreserveLeadOne in SIMD-fashion incurs $\lceil \log_2(n) \rceil + 2$ multiplications, and $\lceil \log_2(n) \rceil + 1$ rotations.

Matrix-Vector Product. A naive approach for matrix-vector multiplication in SIMD-fashion would be to encrypt the matrix M row-wise, i.e. $[M_i^{\mathsf{row}}] = [M_{i,1}, ..., M_{i,n}]$, for $i \in [n]$, and then evaluate $\mathsf{InnerProd}$ over the product of each matrix row and vector $[M_i^{\mathsf{row}}] \cdot [\boldsymbol{v}]$, resulting in n separate vectors. To obtain a single vector that encodes $[M\boldsymbol{v}]$ (which we require to update the adjacency matrix in $\Pi_{\mathsf{TTC\text{-}SIMD}}$), it becomes necesssary to individually extract the first element of $\mathsf{InnerProd}([M_i^{\mathsf{row}}], [\boldsymbol{v}])$ in each encryption of row i. This results in n multiplications, n rotations and a multiplicative depth of 2.

Instead, we avoid extraction and keep the multiplicative depth to 1 by implementing a technique by Halevi and Shoup [HS14], which encodes the matrix *diagonals* and results a single vector ($[M\boldsymbol{v}]$) after n multiplications and rotations, while maintaining multiplicative depth of 1 (see Figure 13).

Let $\mathsf{diag}_l(M)$ denote the l'th diagonal of $n \times n$ matrix M in vector form. Concretely, let the element at index $i \in [n]$ of the l'th diagonal be defined as

$$\mathsf{diag}_l(M)[i] = M_{i,(l+i)\%(n)} \tag{7}$$

A matrix-vector multiplication is then evaluated in SIMD-fashion with a single (encrypted) output vector as follows;

$$[M\boldsymbol{v}] = \sum_{0 \le l < n} [\mathsf{diag}_l(M)] \cdot \mathsf{rot}([\boldsymbol{v}], l, \mathsf{left}) \tag{8}$$

In our implementation, the matrix will be the encoding of a client's priority list as a permutation matrix (and its transpose). Thus, at the start of the protocol, clients encode their preferences as permutation matrix diagonals. For an F_{ABB} parameterized to operate over vectors of n^2, as in $\Pi_{\mathsf{TTC\text{-}SIMD}}$ in Figure 6, we note that the vector $\boldsymbol{v}$ can be replicated to the right, namely $[\boldsymbol{v}, \boldsymbol{v}, 0, ..., 0]$ to maintain correctness of Eq. G.3 over rotation operations.

Matrix-Matrix Product. We implement a technique by Jiang et al. [JKLS18] for multiplication of $n \times n$-matrices with multiplicative complexity $O(n)$ (see Figure 13). We sketch the main ideas below and refer the readers to [JKLS18] for details on correctness. Matrices are encoded in flattened form as a single vector by arranging all matrix rows in the order of their row index;

$$\mathsf{flat}(M) = \boldsymbol{m} = (M_1^{\mathsf{row}}, ..., M_n^{\mathsf{row}})$$

Subsequent SIMD operations are thus over vectors of n^2 length. In [JKLS18], the product of matrices A and B is considered as the sum of n element-wise products of $\mathsf{colShift}_i(\mathsf{lin}(\boldsymbol{a}))$ and $\mathsf{rowShift}_i(\mathsf{lin}'(\boldsymbol{b}))$ for $i \in [n]$, where lin and lin' represent the following linear transformations on A and B in flat-encoded form.

– $\mathsf{lin}(\boldsymbol{a}) = \sum_{-n<k<n}(\boldsymbol{u}_k \cdot \mathsf{rot}(\boldsymbol{a}, k))$
  where $\boldsymbol{u}_{-n<k<0}[l] = \begin{cases} 1 & \text{if } -k \leq l - (n+k) \cdot n < n \\ 0 & \text{otherwise} \end{cases}$ and $\boldsymbol{u}_{0\leq k<n}[l] = \begin{cases} 1 & 0 \leq l - n \cdot k < (n-k) \\ 0 & \text{otherwise} \end{cases}$
– $\mathsf{lin}'(\boldsymbol{b}) = \sum_{0\leq k<n}(\boldsymbol{u}'_k \cdot \mathsf{rot}(\boldsymbol{a}, k))$
  where $\boldsymbol{u}'_{0\leq k<n}[l] = \begin{cases} 1 & \text{if } l = k + n \cdot i \text{ for } 0 \leq i < n \\ 0 & \text{otherwise} \end{cases}$

The column and row shifting of flat-encoded matrix $\mathsf{flat}(M) = \boldsymbol{m}$ can be expressed as

– $\mathsf{colShift}_i(\boldsymbol{m}) = \boldsymbol{v}_i \cdot \mathsf{rot}(\boldsymbol{m}, i) + \boldsymbol{v}_{i-n} \cdot \mathsf{rot}(\boldsymbol{m}, i-d)$
  where $\boldsymbol{v}_i[l] = \begin{cases} 1 & \text{if } 0 \leq l \mod n < (n-i) \\ 0 & \text{otherwise} \end{cases}$ and $\boldsymbol{v}_{i-n}[l] = \begin{cases} 1 & (n-i) \leq l \mod n < n \\ 0 & \text{otherwise} \end{cases}$
– $\mathsf{rowShift}_i(\boldsymbol{m}) = \mathsf{rot}(\boldsymbol{m}, i \cdot n)$

Thus, the flat-encoded of the product of A and B is evaluated over their flat-encoded form as follows, again over vectors of length n^2.

$$[\,\mathsf{flat}(AB)\,] = \sum_{i\in[n-1]} \mathsf{colShift}_i(\mathsf{lin}([\boldsymbol{a}])) \cdot \mathsf{rowShift}_i(\mathsf{lin}'([\boldsymbol{b}])) \tag{9}$$

Matrix-matrix multiplication over flat-encoded matrices in vectors of n^2 require $\sim 6n$ homomorphic multiplications and rotations (and a multiplicative depth of 3).

For an overview of how the various SIMD-algorithms described in this section are integrated in $\Pi_{\mathsf{TTC\text{-}SIMD}}$ (Figure 6), we refer the reader to Appendix G.

	Additions	Rotations	Multiplications	Mult. Depth
Matrix-Vector Mult. [HS14]	$n-1$	n	n	1
Matrix-Matrix Mult. [JKLS18]	$2n-2$	$6n-4$	$6n-4$	3

Fig. 13: Complexity of SIMD-style matrix operations.

	Additions	Rotations	Multiplications	Mult. Depth
NotEqZero	$n+1$		n	$\lceil \log_2(n+1) \rceil$
PrefixAdd	$\lceil \log_2(n) \rceil$	$\lceil \log_2(n) \rceil$		
PrefixMult		$\lceil \log_2(n) \rceil$	$\lceil \log_2(n) \rceil$	$\lceil \log_2(n) \rceil$
InnerProduct	$\lceil \log_2(n) \rceil$	$\lceil \log_2(n) \rceil$	2	
PreserveLeadOne		$\lceil \log_2(n) \rceil + 1$	$\lceil \log_2(n) \rceil + 2$	$\lceil \log_2(n) \rceil + 2$

Fig. 14: Complexity of SIMD-style tasks in $\Pi_{\mathsf{TTC\text{-}SIMD}}$.

H Refreshing intervals & comparison to secret-sharing based MPC

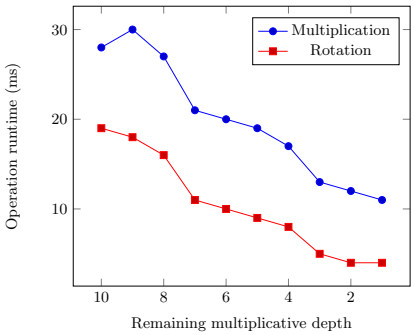

Fig. 15: BGV parameterized with $p = 65537$ and multiplicative depth 10. Homomorphic multiplication and rotation runtimes with decreasing remaining circuit depth.

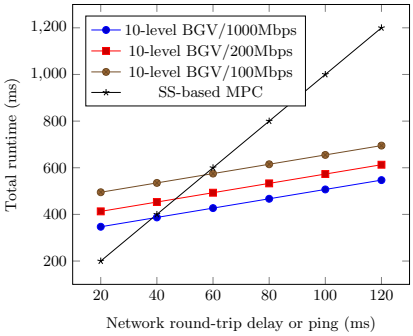

Fig. 16: Estimated 10-depth circuit evaluation runtimes with leveled HE (BGV) and secret-sharing based MPC.

Refreshing intervals. Choosing the correct maximal depth for the encryption scheme we use, is a non-trivial task. Choosing a bigger maximal depth significantly affects the computational over-head of performing a single homomorphic multiplication, whereas choosing a smaller maximal multiplicative depth, would increase the number of rounds of interaction between the involved parties. Finding an appropriate maximal depth essentially requires finding a good balance between computational overheads and round-trip latencies.

Even for a fixed maximal multiplicative depth, the computational cost of a homomorphic multiplication (or rotation) depends on the depth at which it is performed. The more multiplicative depth has been "consumed", the faster the homomorphic multiplications become, as is illustrated in Figure 15.

For our parameterization of BGV, we observe an average multiplication runtime of 21 ms over our 11 multiplicative levels, resulting in a computational

runtime of 222 ms between ciphertext refreshes. Here, the final multiplicative level is required for consolidating different ciphertexts into a single one for more efficient transmission. Our estimated runtimes for a 10-depth circuit with BGV including ciphertext refreshing over various network conditions are shown in Figure 16; the estimated communication component incurred during refreshing is obtained by subtracting the purely computational runtime of 222 ms. The detailed assumptions behind this estimated communication overhead during refreshing of BGV ciphertexts follow below.

Comparison to secret-sharing based MPC. Given that our implementation fixes the multiplicative depth to be a constant, strictly speaking in asymptotic terms, our communication overhead for refreshing ciphertexts grows linearly with the multiplicative depth of the evaluated circuit. Nonetheless, we argue that practically speaking our approach using SHE is superior to a secret-sharing based approach under realistic network conditions.

In Figure 16, we illustrate estimated runtimes for the evaluation of an arithmetic circuit with a multiplicative depth of 10 using both our leveled-HE instantiation and a generic secret-sharing-based MPC. The figure examines how the total runtimes of the two approaches are affected by different network conditions. Concretely, we estimate the *network latency = round-trip-delay + transmission delay*, where round-trip-delay corresponds to "ping" times and the transmission delay is given by *data size / transmission rate*. For the sake of simplicity, we assume that all multiplications at the same depth level can be performed perfectly in parallel in both approaches.

For secret-sharing based MPC, we only measure the time induced by network latency, ignoring the transmission delay and assuming that local computations take no time at all. These assumption heavily favours the secret-sharing based approach.

For the leveled-HE approach, we use the same plaintext modulus $p = 65537$ as above and set the multiplicative depth to be 11. We measure the local computation times as well as the transmission delay. We assume that each round of communication requires sending a single ciphertext. This can be achieved by locally packing all individual ciphertexts into one, assuming that sufficiently many slots are available, which there are for all of our settings. The packing itself can be done via one layer of rotations and some additions, which we conservatively approximate with the cost of an additional layer of multiplications. For our parameters, the ciphertext that is sent over the wire is of size 514KB .

As an exemplary data point from Figure 16, let us consider a 200 Mbit/s = 25 MB/s transmission rate. We estimate 0.041 s = 514 KB/$(25 \cdot 10^3/2$ KBps) transmission time from the HE evaluation server to the two other servers . Assuming network round-trip delay $\Delta = 40$ ms, a ciphertext refresh will incur $25 + 35$ ms in de-/encryption runtime, and $2 \cdot (40 + 41)$ ms total network latency for both the transmission of ciphertext copies to the servers and the receipt of decryption shares, resulting in a ciphertext refresh runtime of 222 ms. Given that one level of multiplications takes 21 ms on average, our estimated total runtime

for leveled HE is $453 = 231 + 222$ ms for the evaluation of an arithmetic circuit of depth 10 and the subsequent ciphertext refresh.

I Upgrading to malicious security

In this section, we review the changes we would need in our protocols to get malicious security and how they might affect efficiency. Our protocol for implementing TTC based on F_{ABB} is already malicious secure, as it only consists of a sequence of calls to F_{ABB} that all parties must agree on. Thus, the only change we need is to verify the input that clients send. The input is given in the form of a set of ciphertexts, so clients need to prove in zero-knowledge that they know the corresponding plaintexts. Such proofs of plaintext knowledge have been studied (and implemented) extensively in the literature (see [DPSZ12], for instance), and they would add only a constant factor overhead. We also need to verify that a client supplies a permutation matrix P and its inverse P^{-1}. First, we can check that all entries in (the encryption of $)P$ are 0 or 1 using the standard trick of checking that they are all roots of the polynomial $X(X-1)$. Then we can sum all rows and all columns and open to check that the sums are all 1, which implies that all rows and columns contain exactly one 1, i.e., P is a permutation matrix. And finally we can transpose P, subtract from the alleged P^{-1} and open to check that the result is 0. This all adds an overhead linear in the size of the input and is anyway only needed for the inputs, not for the intermediate results of the computation.

The major part of the protocol itself is a computation on ciphertexts. This is a computation that all parties can do to make sure there is agreement on what to decrypt. Although this is more computation than the semi-honest case where only one party needs to compute, it will not add much to the wall-clock time, as all parties can do it in parallel.

Some steps of the semi-honest protocol creates random ciphertexts by adding a contribution from each party, in the malicious case we can do the same, but we need to add proofs of plaintext knowledge, as discussed above.

Finally, the standard decryption protocol for BGV from [DPSZ12] is only semi-honest secure. However, in [DPR16] a technique based on algebraic manipulation detection codes is shown, that allows to verify the result of a decryption and abort if a problem is found (since we are doing dishonest majority, aborting in case of an error is the only option anyway). This technique needs a random ciphertext, one extra multiplication on ciphertext and two additional decryptions.

In conclusion, we believe that upgrading to malicious security is relatively straightforward and would not lead to a prohibitive overhead,