Prime Match: A Privacy-Preserving Inventory Matching System

Antigoni Polychroniadou *Gilad Asharov*[1] *Benjamin Diamond*[2] *Tucker Balch*
Hans Buehler[2] *Richard Hua* *Suwen Gu* *Greg Gimler*[2] *Manuela Veloso*

J.P. Morgan

Abstract

Inventory matching is a standard mechanism for trading financial stocks by which buyers and sellers can be paired. In the financial world, banks often undertake the task of finding such matches between their clients. The related stocks can be traded without adversely impacting the market price for either client. If matches between clients are found, the bank can offer the trade at advantageous rates. If no match is found, the parties have to buy or sell the stock in the public market, which introduces additional costs.

A problem with the process as it is presently conducted is that the involved parties must share their order to buy or sell a particular stock, along with the intended quantity (number of shares), to the bank. Clients worry that if this information were to "leak" somehow, then other market participants would become aware of their intentions and thus cause the price to move adversely against them before their transaction finalizes.

We provide a solution that enables clients to match their orders efficiently with reduced market impact while maintaining privacy. In the case where there are no matches, no information is revealed. Our main cryptographic innovation is a two-round secure *linear* comparison protocol for computing the minimum between two quantities without preprocessing and with malicious security, which can be of independent interest. We report benchmarks of our Prime Match system, which runs in production and is adopted by a large bank in the US – J.P. Morgan. Prime Match is the first secure multiparty computation solution running live in the financial world.

1 Introduction

An axe is an interest in a particular stock that an investment firm wishes to buy or sell. Banks and brokerages provide their clients with a matching service, referred to as "axe matching". When a bank finds two clients interested in the same stock but with opposite directions (one is interested in buying and the other is interested in selling), the bank can offer these two clients the opportunity to trade internally without impacting the market price. Both clients, and the bank, benefit from this internalization. On the other hand, if the bank cannot find two matching clients, the bank has to perform the trade in the public market, which introduces some additional costs and might impact the price. Banks, therefore, put efforts into locating internalized matches.

One such effort is the following service. To incentivize clients to trade, banks publish a list of stocks that they are interested in trading, known as "axe list". The axe list that the bank publishes contains, among other things, aggregated information on previous transactions that were made by clients and facilitated by the bank. For instance, to facilitate clients' trades, the bank sometimes buys stocks that some clients wish to sell. The bank then looks to sell those stocks to other clients at advantageous rates before selling those stocks in the public market. Those stocks will appear in the bank's axe list.

The axe list consists of tuples $(\mathsf{op}, \mathsf{symb}, \mathsf{axe})$ where $\mathsf{op} \in \{\mathsf{buy}, \mathsf{sell}\}$, symb is the symbol of the security to buy or sell, and axe is the number of the shares (quantity) of the security to buy or sell (we sometimes use the terminology of "long" for buy and "short" for sell). This axe list provides clients the ability to locate available (synthetic) trades at reduced financing rates.

Unfortunately, this method is unsatisfactory. Although the information in the axe list of the bank relates to transactions that were already executed, there is a correlation between previous transactions that a client performed and future transactions that it might wish to trade. Thus, clients feel uncomfortable with seeing their recent (potentially large) trade history (although anonymized and aggregated) in the axe list that the bank publishes, and sometimes ask the bank to remove their previous trades from the axe list. Clients, therefore, face the following dilemma: keeping their axes published reveal information about their future potential trades or investment strategy, while continuously asking to remove trades from the

[1]Department of Computer Science, Bar-Ilan University. Work initiated while at J.P. Morgan AI Research. Editorial work while at Bar-Ilan University, supported by J.P. Morgan Faculty Research Award.

[2]Authors contributed while employed at J.P. Morgan. Buehler moved to XTX Markets, Diamond to Coinbase, and Gimler to Meta.

axe list limits the banks' ability to internalize trades and offer advantageous rates, to begin with.

The bank currently uses some ad-hoc methods to mitigate the leakage. For instance, it might aggregate several stocks together into "buckets" (e.g., reveal only the range of available stocks to trade in some sector), or trim the volumes of other stocks. This does not guarantee privacy, and also makes it harder to locate potential matches.

1.1 Our Work

We provide a novel method for addressing the inventory matching problem (a simple double auction, which is periodic, with a single fixed price per symbol). Our main contribution is a suite of cryptographic protocols for several variants of the inventory matching problem. The system we report, called Prime Match, was implemented and runs in production in J.P. Morgan since September 2021. Prime Match has the potential to transform common procedures in the financial world. We design the following systems:

- *Bank-to-client inventory matching*: Prime Match supports a secure two-party (bank-to-client) inventory matching. The client can privately find stocks to trade in the bank's full axe list without the bank revealing its axe list, and without the client revealing the stocks and quantities it wishes to trade. The protocol is secure against a semi-honest bank and malicious client and is of two rounds of communication (three rounds if both parties learn the output).
- *Client-to-client inventory matching*: We extend Prime Match to support a secure (client-to-client) inventory matching. This is a three-party protocol where the bank is an intermediate party that mainly delivers the messages between two clients and facilitates the trade if there is a match. This enables two clients to explore whether they can have potential matches against each other and not just against the axe list of the bank. This further increases potential matches. The protocol is secure in the presence of one malicious corruption and is of three rounds of interaction.
- *Multiparty inventory matching:* We also extend the client-to-client inventory matching to multiple clients coming at once and looking to be matched.

We expand on each one of those scenarios below.

Bank-to-client inventory matching: We replace the current procedure in which the bank sends an axe list to a client, and the client replies with which stocks to trade based on the axe list, with a novel bank-to-client inventory matching. Prime Match allows the bank to locate potential matches without revealing its axe list, and without the client revealing its interests. Moreover, as the bank can freely use accurate axe information (as the axe list is hidden), clients have no longer an interest to remove themselves from the axe list. All parties enjoy better internalization and advantageous rates.

Importantly, the bank does not learn any information about what the client is interested in on any stock that is not matched, and likewise, the client does not learn any information on what is available unless she/he is interested in that as well. Only after matches are found, the bank and the client are notified and the joint interest is revealed. At a high level, for two orders $(\mathsf{buy}, \mathtt{X}, \mathsf{axe}_1)$ and $(\mathsf{sell}, \mathtt{X}, \mathsf{axe}_2)$ on the same symbol $\mathtt{X}$, we provide a secure two-party protocol that computes as the matching quantity the min quantity between axe_1 and axe_2.[1]

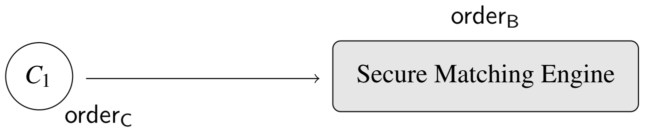


Figure 1: Client-to-bank topology. Client C sends an encrypted order $\mathsf{order}_C = (\mathsf{buy}, \mathtt{X}, \mathsf{axe}_1)$ to the Bank (secure matching engine) which holds $\mathsf{order}_B = (\mathsf{sell}, \mathtt{X}, \mathsf{axe}_2)$. The engine computes the minimum between axe_1 and axe_2.

Client-to-client inventory matching: The above approach only enables matching between the bank's inventory to each client separately but does not allow a direct matching among different clients. For illustration, consider the following scenario: Client A is interested in buying 100 shares of some security $\mathtt{X}$, while client B is interested in selling 200 shares of the same security $\mathtt{X}$. On the other hand, the bank does not provide $\mathtt{X}$ in its inventory axe list. The bank either distributes in a non-private way its axe list to clients A and B (as it is being conducted prior to our work) or engages twice in a bank-to-client inventory matching described above, the first time against client A and the second time against client B. The two clients do not find $\mathtt{X}$ in the list, and both clients would have to trade on the public market at higher costs.

Prime Match allows the clients and the bank not to miss such opportunities. We provide a mechanism that acts as a transparent matching engine. Each client provides as input to the computation his/her encrypted axes, and the clients then interact and learn whether their axes match or not, see Figure 2. For this solution, we provide a three-party secure minimum protocol Π_{min} among two clients and the bank as the intermediary party to facilitate and execute the trade if there is a match.

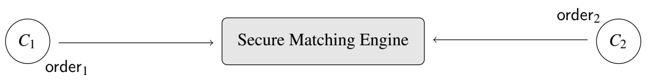


Figure 2: Client-to-client topology. Client C_1 and C_2 send encrypted orders $\mathsf{order}_1 = (\mathsf{buy}, \mathtt{X}, \mathsf{axe}_1)$ and $\mathsf{order}_2 = (\mathsf{sell}, \mathtt{X}, \mathsf{axe}_2)$, respectively, to the Bank which computes the minimum of axe_1 and axe_2.

[1]In our actual protocol, each party also provides a range of quantities it wishes to trade, i.e., a minimum amount and a maximum amount. If there is no match that satisfies at least its minimum quantity, then there is no trade. To keep the introduction simple, we omit this additional complexity for now.

Multi-party protocol. A potentially powerful mechanism would be to support multiple clients coming at the same time, where all clients talk to each other through the bank who facilitates the trades when there are matches. This might increase the potential number of matches for each client. We implement such a mechanism based on our client-to-client matching protocol, where we invoke it $\binom{n}{2}$ times, for each possible pair of clients, when n is the total number of participating clients. Since the service of axe matching is relatively exclusive, i.e., it is offered only to selected clients, n is relatively small (around 10 per day), and thus this approach suffices for the current needs.

At this point, we only implement this relatively degenerated form of multiparty matching. We provide security for a semi-honest bank and malicious clients. In the multiparty setting, there are further challenges that have to be explored, such as what information is leaked by the functionality due to partial matches (i.e., client A can fulfill its order, say, selling 1000 shares by matching with client B and C, each wishing to buy 500 shares). Moreover, to achieve malicious security, the protocol also has to guarantee that the bank does not discriminate against clients, e.g., when two clients are both interested in buying some security, then it treats them fairly and does not prefer to match the "big client" over the "small client." In fact, it is impossible to support security against a malicious bank in this case already because of the star network – the clients communicate through the bank with no authentication (see [7]). Therefore, achieving malicious security would require some different setups and further techniques. We leave this for future research.

From a business perspective, the clients generally do trust the bank, and the bank is also highly regulated and will not risk its reputation by attempting to cheat. Therefore, semi-honest bank generally suffices.

Secure minimum protocol. At the heart of our Prime Match engine is a secure protocol for comparing two input values axe_1 and axe_2, each in $\{0,\ldots,2^n-1\} \subset \mathbb{F}_q$. The protocol, given the bit-decompositions of axe_1 and axe_2, computes the minimum between the two. We have a two-party variant (bank to client) and a three-party variant when only two parties have inputs (client-to-client) and the third party (the server) helps in the computation. For the latter, an interesting property of our protocol is that the two clients only perform linear operations, and therefore can operate non-interactively on encrypted inputs (or secret-shared, or homomorphic commitments, etc.). The server facilitates the computation. For ℓ-bit inputs, our protocol runs in three rounds of interaction and with $O(\ell^2)$ communication where in the first round clients provide their input, and in the last (third) round the output is revealed. The protocls also offers malicious security.

Implementation and evaluation. All three scenarios were implemented, and we report running times in Section 5. On the bottom level, both bank-to-client and client-to-client protocols can process roughly 10 symbols per second with security against malicious clients under conventional machines with commodity hardware. Our system is running live, in production by J.P. Morgan. To the best of our knowledge, this is the first MPC solution running live in the financial world. Commercially, the main advantage of the system is the increased opportunities for clients to find matches.

As clients do not wish to spend resources to use such a service (installation of packages, maintenance cost, etc.), and cannot commit to providing tech resources before testing the product, Prime Match is implemented as a browser service. This raises several challenges in the implementation, see Section 5. Moreover, in the client-to-client matching a star topology network is required where clients communicate only with the bank. Clients do not wish to establish communication with other clients and reveal their identities to other clients.

Our contributions. To conclude, our contributions are:

- We identify a real-world problem in which cryptography significantly simplifies and improves the current inventory matching procedure.
- We provide two new protocols: bank-to-client inventory matching and client-to-client inventory matching. Those completely replace the current method which leaks information and misses potential matches. Our protocols are novel and are specifically tailored to the problem at hand. We do not just use generic, off-the-shelf, MPC protocols (see Section 1.3 for a discussion).
- At the heart of our matching engine is a novel two-round comparison protocol that minimizes interaction and requires only linear operations.
- The protocols are implemented and run live, in production, by a major bank in the US – J.P. Morgan.

1.2 Related Work

Prior works on volume (quantity) matching. We now compare the prior privacy-preserving volume matching architectures [4,9,13] to Prime Match. The MPC-based volume matching constructions of [9,13] derive their security by separating the system's *service operator/provider* into several (e.g., 3) distinct servers, whose collusion would void the system's security guarantees. The clients submit their *encrypted* orders to the servers by secret sharing, such that no single server can recover the encrypted orders. The clients have no control over these servers and no clear way to prevent them from colluding.

Allowing clients to *themselves* serve as contributing operators of the system would present its own challenges. For instance, it would impose a disproportionate computational burden on those clients who choose to serve as operators. Moreover, it is unclear how to incentivize clients to run heavy computations, and to play the role of the operators.[2]

The fully homomorphic approach of [6] imposes a computational burden on a single server in a star topology network in which clients communicate with the server. Moreover, the concrete efficiency of the proposed GPU-FHE scheme is slow. Furthermore, the scheme of [6] does not offer malicious security. FHE-based solutions for malicious security are much less efficient than the ones based on MPC.

Prior works on privacy-preserving dark pools. A recent line of research has attempted to protect the information contained in dark pools [4, 10] run by an operator. The systems described in these works allow users to submit orders in an "encrypted" form; the markets' operator then compares orders "through the encryptions", unveiling them only if matches occur. The functionality of privacy-preserving dark pools is a continuous double auction in which apart from the *direction* (buy or sell) and a desired trading *volume*, a *price* (indicating the "worst" price at which the participant would accept an exchange) is submitted. The operator "matches" compatible orders, which, by definition, have opposite directions, and for which the price of the buy order (the "bid") is at least the price of the sell order (the "ask"). [9, 10] are based on MPC with multiple operators and the work of [4] is based on FHE.

Dark pools are different than our setting, as matches are also conditioned on an agreement on a price (requiring many more comparisons) leading to more complex functionality. In comparison, inventory matching is a simple double auction, which is periodic, with a single fixed price per symbol. Moreover, dark pools support high-frequency trading, which means that they have to process orders very fast. All prior works' performance on dark pools (including multi-server dark pools) does not suffice for high-frequency trading. In comparison, axe-list matching is a much slower process; with the current, insecure procedure of axe-matching, a few minutes might elapse between when the bank sends its axe list, and the time the client submits its orders. Since ensuring privacy introduces some overhead, clients might not necessarily prefer a slower privacy-preserving dark pool over a fast ordinary dark pool. Furthermore, secure comparison is a necessary building block for dark pools. Any of the comparison protocols from prior works, [11, 14, 23, 24, 27, 28], including ours, can be used for dark pools, but all of them have some overhead. Unfortunately, neither of these works can lead to a fast dark pool (in a star topology network) which is close to the running times of a dark pool operating on plaintexts. Achieving fast enough comparison that is suitable for high-frequency trading is an interesting open problem.

The work of Massacci et al. [25] considers a distributed Market Exchange for futures assets which has functionality with multiple steps where one of the steps includes the dark pool functionality. Their experiments show that their system can handle up to 10 traders. Moreover, orders are not concealed: in particular, an aggregated list of all waiting buy and sell orders is revealed which is not the case in solution and the dark pool solutions. Note that there are works that propose dark pool constructions on the blockchain [5, 19, 26] which is not the focus of our work. Moreover, these solutions have different guarantees and security goals. None of the above solutions is in production.

Prior works on secure 3-party Less Than comparison. There are several works in the literature that propose secure comparison protocols of two values in the information-theoretic setting [1, 11, 14, 23, 24, 27, 28]. See Table 1 for a detailed comparison of these works compared to ours. Our protocol does not require preprocessing and runs in 2 rounds of interaction. Our cost incurs an ℓ^2 overhead since we secret share ℓ bit numbers in a field of size ℓ. Similar overhead also appears in prior works. The security parameter λ overhead is required due to the use of coin flipping and the additional use of commitments in the malicious protocol. The main reason for the higher overhead of prior secret sharing-based protocols in Table 1 is that they require interaction per secure multiplication leading to an increased round complexity ($\approx \log \ell$). Our protocol does not require any secure multiplications, which is a significant benefit in upgrading our passive protocol to one with malicious security.

The works of [15, 16, 18, 22], based on multiplicative/additive homomorphic encryption, provide 2 (or constant) round solutions but they only offer passive security. The computational cost is capped at $O(\lambda \cdot \ell)$ modular multiplications. Moreover, some works require a trusted setup assumption to generate the public parameters. For instance the modulus generation of the homomorphic Paillier encryption-based solutions.

The most recent work of [1], based on functional secret sharing in the preprocessing model, is a three-round solution offering only passive security with the cost of $O(\ell)$ PRG calls in its online phase.

1.3 Why Specifically-Tailored Protocols?

A natural question is why we design a specifically tailored protocol for the system, instead of just using any generic, off-the-shelf secure computation protocols. Those solutions are based on securely emulating arithmetic or Boolean circuits, and require translating the problem at hand to such a circuit. Specifically, for our client-to-client matching algorithm, which is a three-party secure protocol with one corruption, it looks promising to use some generic MPC protocols that are based on replicated secret sharing, such as [2, 12] or garbled circuits [21, 29].

There are two main requirements from the system (from a business perspective) that leads us to design a specifically-tailored protocol and not a generic MPC: (1) The need for a constant number of rounds; (2) Working with committed

[2]Part of the success of the Prime Match system is related to the fact that clients are offered a web service to participate in the system which requires minimal tech support by the clients.

Protocol	Offline Communication	Online Communication	Offline Computation	Online Computation	Rounds	Security	Corruption
[27]	-	$O\big((\ell\log\ell)\cdot(\ell+s)\big)$	-	$O\big((\ell\log\ell)\cdot(\ell+s)\big)$	31	passive	HM
[11]	-	$O\big(\ell\cdot(\ell+s)\big)$	-	$O\big(\ell\cdot(\ell+s)\big)$	$O(\log\ell)$	passive	HM
[28]	-	$O(\ell^2+\log\ell)$	-	$O(\ell^2+\log\ell)$	$O(\log\ell)$	passive	DM
This work	-	$O(\ell^2+\log\ell)$	-	$O(\ell^2+\log\ell)$	2	passive	DM
[23]	$O(\ell^2)$	$O(\log\ell\cdot(\ell+s))$	$O(\ell^2)$	$O(\log\ell\cdot(\ell+s))$	$O(\log\ell)$	active	HM
[14]	-	$O\big((\ell\log\ell)\cdot(\ell+s)\big)$	-	$O\big((\ell\log\ell)\cdot(\ell+s)\big)$	44	active	HM
[24]	$O(\ell)$	$O\big((\ell\log\ell)\cdot(\ell+s)\big)$	$O(\ell)$	$O\big(\ell\cdot(\ell+s)\big)$	$O(\log\ell)$	active	DM
This work	-	$O(\ell\cdot(\ell+\lambda))$	-	$O(\ell\cdot(\ell+\lambda))$	2	active	HM

Table 1: Cost of passive and active comparison protocols in terms of offline, and online communication and computation complexity; in terms of rounds; in terms of security; and in terms of corruptions supported. HM stands for honest majority, while DM stands for dishonest majority. ℓ denotes to the bit length of the input, s is the statistical security parameter and λ is the computational security parameter. The work of [24] achieves statistical security over arithmetic fields but it achieves perfect security over the arithmetic rings.

inputs. Furthermore, no offline preprocessing is possible since clients wish to participate only during the live matching phase. We provide a comparison with generic MPC techniques in the full version.

1.4 Overview of our Techniques

We focus in this overview on the task of client-to-client matching (see Figure 2): A three-party computation between two clients that communicate through the bank. We present our solution while hiding only the clients' quantities axe. However, our detailed protocol additionally hides both the directions and the symbols. We present our protocol in the semi-honest setting and then explain how to achieve malicious corruption.

Semi-honest clients and server: The client provides secret shares (and commitments) for all possible symbols and for the two possible sides. If a client is not interested in buying (resp. selling) a particular stock, it provides 0 as its input for that symbol and size. It is assumed that the total number of symbols is around $1000-5000$, and of course, the number of sides is 2. Thus, each party has to provide roughly $2000-10000$ values. To see if there is a match between clients A and B on a particular stock, we securely compute the minimum between the values the parties provided with opposite sides (i.e., A sells and B buys, or B sells and A buys).

Each one of the clients first secret shares its secret value axe using an additive secret sharing scheme. The two clients then exchange shares[3]. Then, they decide on the matching quantity by computing two bits indicating whether the two quantities are equal or which one of the two is minimal.

We design a novel algorithm for computing the minimum. The algorithm consists of two phases. As depicted in Figure 3, the first phase works on the shares of the two secrets (a,b), exchanged via the matching engine using symmetric key encryption, while performing only linear operations on them (min protocol). Looking ahead, each one of the two clients would run this phase, without any interaction, on its respective shares. The result would be shares of some secret state $(\mathsf{d}_0,\mathsf{d}_1)$ in which some additional non-linear processing is needed after reconstruction to obtain the final result. However, the secret state can be simulated with just the result of the computation – i.e., the two bits indicating whether the two numbers are equal or which one is minimal. Therefore, at the end of the first phase, the two clients can send the shares to the server, who reconstructs the secret state and learns the result, again using just local (this time, non-linear) computations.

Our minimum protocol min is described in Section 4.2, and we overview our techniques and contributions in the relevant section. Our semi-honest protocol is given in Section 4.3.

Malicious clients. We now discuss how to change the protocol to protect against malicious clients.

Zooming out from computing minimum, the auction works in two phases: a "registration" stage, where clients submit their orders, and the matching stage, where the clients and the bank run the secure protocols to find matching orders. In the malicious case, the parties submit commitments to the quantities of their orders to the server. The list of participants is not known in advance, only the clients who submitted a commitment can participate in the current matching phase. Moreover, the list of participants (at each run) is not public and is only known to the server.

Of course, clients have to be consistent, and cannot use different values in the matching phase and in the registration phase. In the matching phase, the clients secret shares their inputs (additive secret sharing), and prove using a Zero-Knowledge (ZK) proof that the shares define the committed value provided in the registration phase.

More specifically, client C_1 commits to a during registration, i.e., sends $\mathsf{Com}(\mathsf{a})$ to the server and commits to the

[3] The communication model does not allow the two clients to talk directly, and each client talks only to the server. However, using encryption and authentication schemes, the two clients can establish a secure channel while the server just delivers messages for them.

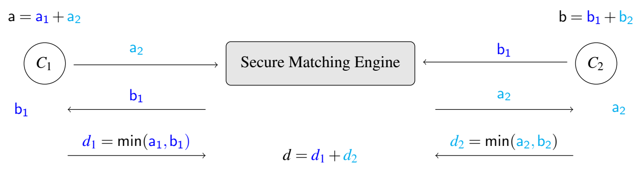
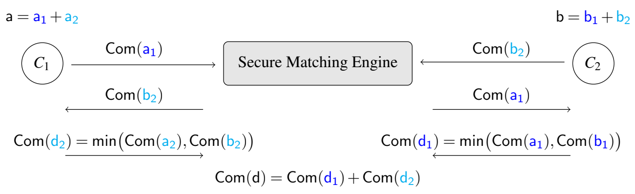


Figure 3: Client-to-client matching protocol for computing the minimum between the quantities a from client C_1 and b from client C_2 in the semi-honest setting. As described in Footnote 3, the communication between the two clients through the server is encrypted, and so the view of the server in this communication is just d_1, d_2.



$a = a_1 + a_2$
$b = b_1 + b_2$
C_1
$\mathsf{Com}(a_1)$
Secure Matching Engine
$\mathsf{Com}(b_2)$
C_2
$\mathsf{Com}(b_2)$
$\mathsf{Com}(a_1)$
$\mathsf{Com}(d_2) = \min(\mathsf{Com}(a_2), \mathsf{Com}(b_2))$
$\mathsf{Com}(d_1) = \min(\mathsf{Com}(a_1), \mathsf{Com}(b_1))$
$\mathsf{Com}(d) = \mathsf{Com}(d_1) + \mathsf{Com}(d_2)$

Figure 4: Client-to-client matching protocol for computing the minimum in the presence of a malicious adversary. In addition to values computed in Figure 3, the parties compute commitments of the value that the other participant is supposed to send to the server.

shares (a_1, a_2) of the minimum by sending $\mathsf{Com}(a_1)$ and $\mathsf{Com}(a_2)$ to the server. It also proves in ZK the statement that $\mathsf{Com}(a) = \mathsf{Com}(a_1) + \mathsf{Com}(a_2)$ given that the commitment scheme is linearly homomorphic allowing to perform additions on committed values.

On top of the basic semi-honest protocol (as depicted in Figure 3) we also exchange the messages shown in Figure 4 where every party forwards a commitment to the other party for the share that it does not hold. Client C_1 receives a commitment to b_2 and client C_2 receives a commitment to a_1.

Next, recall that our minimum protocol requires only linear work from the clients, and thus it allows to work on any linearly-homomorphic cryptosystem, such as linear secret sharing scheme, linearly homomorphic commitments, linearly homomorphic encryption scheme, and so on. In the semi-honest setting, we used this property to work only on the secret shares. We run the linear algorithm three times in parallel, on different inputs:

1. First, each client simply runs the algorithm on additive shares, just as in the semi-honest solution. This is depicted in Figure 3. Running the algorithm on those shares would result in shares of some secret state that will be delivered to the server. The server reconstructs the state and computes the result from this state.
2. Second, the parties run the algorithm on the commitments of the other party's share. This is depicted in Figure 4. Since the commitment scheme is also linearly-homomorphic, it enables Alice to compute a commitment of what Bob is supposed to send to the server in the first invocation, and vice versa.
3. Third, the parties also compute (again, using only linear operations!) information that allows the server to learn the openings of the other party's commitment. This enables the server to check that all values it received in the first invocations are correct.

Malicious server. Our final system does not provide security against a malicious server, unless the two clients can authenticate themselves to each other, or can talk directly. We show that if the client can communicate to each other, then we can also support malicious server for the comparison protocol.

The server receives shares of some secret states, together with commitments of the secret states. It then reconstructs the secret state and checks for consistency. It then has to perform some non-linear operations on the secret state to learn the result. Applying generic ZK proofs for proving that the non-linear operation was done correctly would increase the overhead of our solution. Luckily, the non-linear operation that the server performs is ZK-friendly, specifically, it is enough to prove in ZK a one-out-of-many proof (i.e., given a vector, proving that one of the elements in the vector is zero; see Theorem 4.4). For this particular language, there exists fast ZK solutions [20]. See Section 4.4 for a description and details.

Organization. The paper is organized as follows. In Section 2 we provide the preliminaries, while some are deferred to the appendices. In Section 3 we provide the main matching engine functionality. In Section 4 we provide our protocol for computing the minimum, including the semi-honest and the malicious versions. In Section 5 we report the system performance and in Appendix C we mention challenges pertaining to the deployment of our system.

2 Preliminaries

Notations. We use PPT as an acronym for probabilistic polynomial time. We use λ to denote the security parameter, and $\mathsf{negl}(\lambda)$ to denote a negligible function (a function that is smaller than any polynomial for sufficiently large λ).

Commitment schemes. A commitment scheme is a pair of probabilistic algorithms $(\mathsf{Gen}, \mathsf{Com})$; given public parameters $\mathsf{params} \leftarrow \mathsf{Gen}(1^\lambda)$ and a message m, $\mathsf{com} := \mathsf{Com}(\mathsf{params}, m; r)$ returns a "commitment" to the message m. To reveal m as an opening of com, the committer simply sends m and r (this is sometimes called "decommitment"). For notational convenience, we often omit params. A commitment scheme is *homomorphic* if, for each params, its message, randomness, and commitment spaces are abelian groups, and the corresponding commitment function is a group homomorphism. We always write message and randomness spaces additively and write commitment spaces multiplicatively. See full version for more details.

Zero-knowledge proofs. We use non-interactive zero-knowledge for three languages. See formal treatment in full version.

- **Commitment equality proof:** Denoted as the relation $\mathcal{R}_{\mathsf{ComEq}}$, the prover convinces the verifier that the two given commitments c_0, c_1 hide the same value.
- **Bit proof:** Denoted as the relation $\mathcal{R}_{\mathsf{BitProof}}$, allows the prover to prove that a commitment c hides a bit, i.e., a value in $\{0,1\}$.
- **Out-of-many proofs.** Denoted as the relation $\mathcal{R}_{\mathsf{OneMany}}$, allows a prover to prove that one of the commitments $V_0, \ldots, V_n$ in the statement is a commitment of 0.

3 The Prime Match Main Functionalities

We now describe our Prime Match inventory matching functionalities. We describe the bank-to-client functionality (Section 3.1), the client-to-client functionality (Section 3.2), and the multi-client system (Section 3.3).

3.1 Bank to Client Matching

This variant is a two-party computation between a bank and a client. The bank tries to find matching orders between its own inventory and each client separately. As mentioned in the introduction, this essentially comes to replace the current procedure of axe-matching as being conducted today, with a privacy-preserving mechanism. Today the bank sends its inventory list to the client who then submits orders to the bank. Note, however, that if the bank runs twice with two different clients, and the bank does not hold some security X, and two clients are interested in X with opposite directions, then such a potential match will not be found.
The functionality proceeds as follows. The client sends to the bank its axe list. This includes the list of securities it is interested in, and for each security whether it is interested in long (buy) or short (sell) exposures, and the quantity. The client sends its own list. The functionality finds whether the bank and the client are interested in the same securities with opposite sides, and in that case, it provides as output the matching orders and the quantity is the minimum between the two amounts.

FUNCTIONALITY 3.1 ($\mathcal{F}_{\mathrm{B2C}}$–Bank-to-client functionality)**.**
The functionality is parameterized by the set of all possible securities to trade, a set U.
Input: The bank P^* inputs lists of orders $(\mathsf{symb}_i^*, \mathsf{side}_i^*, \mathsf{amount}_i^*)$ where $\mathsf{symb}_i \subseteq U$ is the security, $\mathsf{side}_i^* \in \{\mathtt{buy}, \mathtt{sell}\}$ and amount_i^* is an integer. The client sends its list of the same format, $(\mathsf{symb}_i^C, \mathsf{side}_i^C, \mathsf{amount}_i^C)$.
Output: Initialize a list of $\mathsf{Matches}$. For each i, j such that $\mathsf{symb}_i^* = \mathsf{symb}_j^C$ and $\mathsf{side}_i^* \neq \mathsf{side}_j^C$, add $(\mathsf{symb}_i^*, \mathsf{side}_i^*, \mathsf{side}_j^C, \min\{\mathsf{amount}_i^*, \mathsf{amount}_j^C\})$ to M. Output M to both parties.

From a business perspective, it is important to note that the input of the client (and the bank) serves as a "commitment" - if a match is found then it is executed right away.

The functionality resembles a set intersection. In set intersection, if some element is in the input set of some party but not in its output, it can conclude that it does not contain in the other set. Here, if a party does not find a particular symbol in its output although it did provide it as an input, then it is still uncertain whether the other party is not interested in that security, or whether it is interested but with the same side. We show how to implement the functionality in the presence of a malicious client or a semi-honest server in Appendix A.

Bank to multiple clients. In the actual system, the bank has to serve multiple clients. This is implemented by a simple (sequential) composition of the functionality. Specifically, the functionality is now reactive where clients first register that they are interested to participate. The bank then runs Functionality 3.1 with the clients – either according to the basis of first-come-first-served, or some random ordering. We omit the details and exact formalism as they are quite natural given a semi-honest bank.

Range functionalities. In Appendix B, we show a variant of the protocol where each party inputs a *range* in which it is interested and not just one single value. I.e., if a matched order does not satisfy some minimal value, it will not be executed. Since the minimum value does not change throughout the execution, whenever the bank receives 0 as a result of the execution it cannot decide whether the client is not interested in that particular symbol, or whether it is interested – but the matched amount does not satisfy the minimum threshold.

3.2 Client to Client Matching

In this variant, the bank has no input and it tries to find potential matches between clients facilitating two clients that wish to compare their inventories. This is a three-party computation where the bank just facilitates the interaction. It is important to notice that the clients do not know each other, and do not know who they are being paired with. The bank selects the two clients and offers them to be paired.

FUNCTIONALITY 3.2 ($\mathcal{F}_{\mathrm{C2C}}$–Client-to-client functionality)**.**
The functionality is parameterized by the set of all possible securities to trade, a set U. This is a three-party

functionality between two clients, P_1 and P_2, and the bank P^*.

Input: The client P_1 inputs a list of orders $(\mathsf{symb}_i^1, \mathsf{side}_i^1, \mathsf{amount}_i^1)$. The client P_2 inputs a list of orders $(\mathsf{symb}_i^2, \mathsf{side}_i^2, \mathsf{amount}_i^2)$, and the bank has no input.

Output: Initialize a list of Matches. For each i, j such that $\mathsf{symb}_i^1 = \mathsf{symb}_j^2$ and $\mathsf{side}_i^1 \neq \mathsf{side}_j^2$, add $(\mathsf{symb}_i^1, \mathsf{side}_i^1, \mathsf{side}_j^2, \min\{\mathsf{amount}_i^1, \mathsf{amount}_j^2\})$ to M. Output M to all three parties.

In the next section, we show how to implement this functionality in the presence of a malicious client or a malicious server, assuming that the two clients can communicate directly, or have a public-key infrastructure. When the two clients can communicate only through the server and there is no public-key infrastructure (PKI) or any other setup, there is no authentication and the server can impersonate the other client. We therefore cannot hope to achieve malicious security. We achieve security in the presence of a semi-honest server. See a discussion in the next subsection.

3.3 The Multi-Client System

We now proceed to the multiparty auction. Here we have parties that register with their intended lists, and the bank facilitates the orders by pairing the clients according to some random order. The functionality is now reactive; The parties first register, in which they announce that they are willing to participate in the next auction, and they also commit to their orders. In the second phase, the bank selects pairs of clients in a random order to perform client-to-client matching. Looking ahead, typically there are around 10 clients that participate in a given auction.

For simplicity of exposition and to ease readability, we write the functionality as the universe is just a single symbol. Moreover, instead of sending the side explicitly, the client sends two integers L and S, representing its interest in long (buy) or short (sell) exposure, respectively. Rationally, each party would put one of the integers as 0 (as otherwise, it would just pay extra fees to the bank). Generalizing the functionality to deal with many symbols is performed in a natural manner, where the number of total symbols is 1000-5000 in practice. The main functionality can process all the different symbols in parallel.

FUNCTIONALITY 3.3 ($\mathcal{F}_{\mathsf{MC}}$ – Multi-client matching)**.** This is an $n+1$ party functionality between n clients $P_1, \ldots, P_n$ and a bank P^*.

Upon initialization, $\mathcal{F}_{\mathsf{MC}}$ initializes a list $\mathcal{P} = \emptyset$ and two vectors $\mathcal{L}$ and $\mathcal{S}$ of size n, where n bounds the total number of possible clients.

$\mathcal{F}_{\mathsf{MC}}$.**Register**$(P_i, L_i, S_i)$**.** Store $\mathcal{L}[i] = L_i$ and $\mathcal{S}[i] = S_i$ and add i to $\mathcal{P}$. Send to the bank P^* the message $\mathsf{registered}(P_i)$.

$\mathcal{F}_{\mathsf{MC}}$.**Process**$()$.

1. Choose a random ordering O over all pairs of $\mathcal{P}$.
2. For the next pair $(i, j) \in O$ try to match between P_i and P_j (we can assume wlog that always $i \leq j$):
   (a) Compute $M_0 = \min(\mathcal{L}[i], \mathcal{S}[j])$, $b_0^0 = (\mathcal{L}[i] \leq \mathcal{S}[j])$, $b_1^0 = (\mathcal{S}[i] \leq \mathcal{L}[j])$.
   (b) Compute $M_1 = \min(\mathcal{S}[i], \mathcal{L}[j])$, $b_0^1 = (\mathcal{S}[i] \leq \mathcal{L}[j])$ and $b_1^1 = (\mathcal{S}[i] \leq \mathcal{L}[j])$.
   (c) Send $(i, j, M_0, M_1, b_i^0, b_i^1)$ to P_i, and (i, j, M_0, M_1) with $(b_0^0, b_1^0, b_0^1, b_1^1)$ to P^*.
   (d) Update $\mathcal{L}[i] = \mathcal{L}[i] - M_0$ and $\mathcal{S}[j] = \mathcal{S}[j] - M_0$.
   (e) Update $\mathcal{S}[i] = \mathcal{S}[i] - M_1$ and $\mathcal{L}[j] = \mathcal{L}[j] - M_1$.

On malicious server. Our final protocol (see full version) for $\mathcal{F}_{\mathsf{MC}}$ is secure in the presence of a semi-honest P^* (and a malicious client). Inherently, clients communicate through a star network where the bank facilitates the communication. Moreover, we assume no PKI, clients do not know how many clients are registered in the system, and how many clients are participating in the current auction. This can be viewed as “secure computation without authentication", in which case the server can always “split" the communication and disconnect several parties from others (see [7] for a formal treatment).

We prove security in the presence of a semi-honest server. In fact, our protocol achieves a stronger notion of a guarantee than just semi-honest, as in particular, it runs the underlying comparison protocol (a single invocation of a client-to-client matching) which is secure against a malicious server.

Another relaxation that we make is that the ordering of pairs is random, and we do not have a mechanism to enforce it. Note also that the functionality leaks some information to the server; in particular, after finding a match, the bank executes it immediately. The bank can infer information about whether two values equal to 0, and therefore whether a client is not interested in a particular symbol. In contrast, each client just learns whether its value is smaller or equal to the value of the other party, and therefore when it inputs 0 it can never infer whether the other party is interested in that symbol or not.

4 Securely Computing Minimum

A pivotal building block in Prime Match is a secure minimum protocol. In Section 4.1, we review our functionality for computing the minimum. We focus on the case of client-to-client matching with an aiding server. We show how to convert the protocol for two parties in Appendix A.

In Section 4.2 we present the underlying idea for computing the minimum. The algorithm computes the minimum while using only linear operations (looking ahead, those would be computed on shared values) while pushing the non-linear operations on reconstructed data. In Sections 4.3 and 4.4 we show a semi-honest and a malicious protocol for computing the minimum, respectively.

4.1 The Minimum Functionality

After receiving a secret integer from each one of the two parties, the functionality compares them and gives as a result two bits – which indicate which one of the two inputs is smaller than the other, or whether they are equal. It also gives the result to the server.

FUNCTIONALITY 4.1 ($\mathcal{F}_{\mathsf{comp}}$: Server-aided secure minimum functionality)**.**
Consider two players, P_0 and P_1, and a server P^*.

- **Input:** P_0 and P_1 respectively send integers v_0 and v_1 in $\{0,\ldots,2^n-1\}$ to $\mathcal{F}_{\mathsf{comp}}$.
- **Output:** $\mathcal{F}_{\mathsf{comp}}$ sends $b_0 := (v_0 \leq v_1)$ to P_0, $b_1 := (v_1 \leq v_0)$ to P_1, (b_0, b_1) to P^*.

In the rest of this section, we will show how to implement this functionality in the presence of a semi-honest (Section 4.3) and malicious adversary (Section 4.4).[4]

4.2 Affine-Linear Comparison Function

We first describe an abstract algorithm which compares two elements v_0 and v_1 of $\{0,\ldots,2^n-1\} \subset \mathbb{F}_q$, given their bit-decompositions. We separate the algorithm into two parts: ComparisonInitial (Algorithm 1) and ComparisonFinal (Algorithm 2). Both parts do not use any underlying cryptographic primitives.

In the first algorithm (ComparisonInitial), all operations on the bit-decompositions of the two inputs v_0 and v_1 are *linear*. Looking ahead, this will be extremely useful when converting the algorithm into a secure two-party protocol, where v_0 and v_1 are additively shared between the two parties (or also just committed, encrypted under additively homomorphic encryption scheme, etc.). In particular, this part of the protocol can be executed without any interaction, just as the algorithm itself when v_0 and v_1 are given in the clear. The second algorithm (ComparisonFinal) can be computed by a *different* party, given all information in the clear. Looking ahead, this will be executed by the server P^* on the outputs of the first part. This part contains some non-linear operations, however, this part of the algorithm does not have to be translated into a secure protocol.

Overview Algorithm 1 (ComparisonInitial)**.** Our approach is inspired by the algorithm of Wagh, Gupta, and Chandran [28, Alg. 3], which compares a *secret-shared* integer with a *public* integer. (Specifically, its inputs consist of an array of public bits and an array of secret-shared bits.) We extend the algorithm and allow the comparison of two private integers using only linear operations.

We achieve this $\mathbb{F}_q$-linearity in the following way. We fix integers v_0 and v_1 in $\{0,\ldots,2^n-1\}$, with big-endian bit-decompositions given respectively by

$$v_0 = \sum_{j=0}^{n-1} 2^{n-1-j} \cdot v_{0,j}, \quad \text{and} \quad v_1 = \sum_{j=0}^{n-1} 2^{n-1-j} \cdot v_{1,j} \,.$$

We follow the paradigm of [28], whereby, for each $j \in \{0,\ldots,n-1\}$, a quantity w_j is computed which equals 0 if and only if $v_{0,j} = v_{1,j}$. Meanwhile, for each $j \in \{0,\ldots,n-1\}$, we set $c_j := 1 + v_{0,j} - v_{1,j} + \sum_{k<j} w_k$ (we also set $c_n := \sum_{k=0}^{n-1} w_k$, as we discuss below). The crucial observation of [28] is that, for each $j \in \{0,\ldots,n-1\}$, $c_j = 0$ so long as $v_{0,j} < v_{1,j}$ as bits (that is, if $1 + v_{0,j} - v_{1,j} = 0$) *and* the higher bits of v_0 and v_1 agree (inducing the equality $\sum_{k<j} w_k = 0$). By consequence, *some* c_j, for $j \in \{0,\ldots,n-1\}$, must equal 0 whenever $v_0 < v_1$. Similarly, $c_n = 0$ whenever $v_0 = v_1$. In summary, $v_0 \leq v_1$ implies that some $c_j = 0$, for $j \in \{0,\ldots,n\}$.

The main challenge presented by this technique is to ensure that the opposite implication holds; that is, we must prevent the sum $c_j := 1 + v_{0,j} - v_{1,j} + \sum_{k<j} w_k$ from equalling 0 (possibly by overflowing) modulo q—that is, even when $w_k \neq 0$ for some $k < j$—and hence yielding a "false positive" $c_j = 0$, which would falsely assert the inequality $v_0 \leq v_1$. [28] prevents this phenomenon by ensuring that each $w_j \in \{0,1\}$, and choosing $2 + n < q$ (they set $n = 64$ and $q = 67$). In fact, [28] defines $w_j := (v_{0,j} - v_{1,j})^2$. Under this paradigm, $c_j := 1 + v_{0,j} - v_{1,j} + \sum_{k<j} w_k$ is necessarily *non-zero* so long as either $v_{0,j} \geq v_{i,j}$ as bits (so that $1 + v_{0,j} - v_{1,j} > 0$) or *any* $w_k \neq 0$, for $k < j$.

This squaring operation is nonlinear in the bits $v_{0,j}$ and $v_{1,j}$, and so it is unsuitable for our setting. We adopt the following recourse instead, which yields $\mathbb{F}_q$-linearity at the cost of requiring that the number of bits $n \in O(\log q)$ (a mild restriction in practice). The key technique is that we may eliminate the squaring—thereby allowing each w_j to remain in $\{-1,0,1\}$—provided that we multiply each w_j by a suitable public scalar. In fact, it suffices to multiply each (unsquared) difference w_j by 2^{2+j}. In Theorem 4.4 below, we argue that this approach is correct.

Our modifications to [28] also include our computation of the *non-strict* inequality $v_0 \leq v_1$—effected by the extra value c_n—as well as our computation of the opposite non-strict inequality, $v_1 \leq v_0$, in parallel. The latter computation proceeds identically, except uses $-1 + v_{0,j} - v_{1,j} \in \{-2,-1,0\}$ at each bit.

Of course, the intermediate value $v_{0,j} - v_{1,j}$ need only be computed once per iteration of the first loop.

Overview of Algorithm 2 (ComparisonFinal)**.** Note that in Algorithm 1, $w_{\mathsf{accum}} = 0$ as long as $v_{0,j} = v_{1,j}$, and it attains

[4]For ease of presentation, Functionality 4.1 is for the semi-honest version of the protocol; For the malicious case, we will use a slightly different functionality on committed inputs; See full version.

Algorithm 1 ComparisonInitial $((v_{0,0},\dots,v_{0,n-1}), (v_{1,0},\dots,v_{1,n-1}))$

```
1: Assign $w_{\text{accum}} := 0$
2: for $j \in \{0,\dots,n-1\}$ do
3:     Set $c_{0,j} := 1 + v_{0,j} - v_{1,j} + w_{\text{accum}}$.
4:     Set $c_{1,j} := -1 + v_{0,j} - v_{1,j} + w_{\text{accum}}$
5:     Set $w_j := (v_{0,j} - v_{1,j})$ and $w_{\text{accum}} \mathrel{+}= 2^{2+j} \cdot w_j$
6: Set $c_{0,n}$ and $c_{1,n}$ equal to $w_{\text{accum}}$
7: Sample a random permutation $\pi \leftarrow \mathbf{S}_{n+1}$
8: for $j \in \{0,\dots,n\}$ do
9:     Sample random scalars $s_{0,j}$, $s_{1,j}$ from $\mathbb{F}_q \setminus \{0\}$.
10:    Assign $d_{0,j} := s_{0,j} \cdot c_{0,\pi(j)}$,
11:    Assign $d_{1,j} := s_{1,j} \cdot c_{1,\pi(j)}$
12: return $(d_{0,0},\dots,d_{0,n})$ and $(d_{1,0},\dots,d_{1,n})$
```

a non-zero value at the first j for which $v_{0,j} \neq v_{1,j}$. Up to that point, $(c_{0,j}, c_{1,j}) = (1,-1)$. At the first j for which $v_{0,j} \neq v_{1,j}$:

- If $v_0 > v_1$ (i.e., $(v_{0,j}, v_{1,j}) = (1,0)$), then we get that $(c_{0,j}, c_{1,j}) = (2,0)$.
- If $v_0 < v_1$ (i.e., $(v_{0,j}, v_{1,j}) = (0,1)$), then we get that $(c_{0,j}, c_{1,j}) = (0,-2)$.

The algorithm then makes sure that no other value of $c_{b,j'} = 0$, essentially by assigning w_{accum} to be non-zero. If $v_0 = v_1$ then $c_{0,n} = c_{1,n} = 0$. Finally, all the bits $(c_{0,0},\dots,c_{0,n}), (c_{1,0},\dots,c_{1,n})$ are permuted and re-randomized with some random scalars. Observe that if $v_0 > v_1$ then all the values $d_{0,0},\dots,d_{0,n}$ are all non-zero, and one of $d_{1,0},\dots,d_{1,n}$ is zero. If $v_1 \geq v_0$ then exactly one of the values $d_{1,0},\dots,d_{1,n}$ is 0 and all $d_{0,0},\dots,d_{0,n}$ are non-zero.

It is crucial that the vectors $(d_{0,0},\dots,d_{0,n})$ and $(d_{1,0},\dots,d_{1,n})$ do not contain any information on v_0, v_1 other than whether $v_0 \leq v_1$ or $v_1 \leq v_0$. Specifically, these values can easily be simulated given just the two bits $v_0 \leq v_1$ and $v_1 \leq v_0$. Therefore, it is safe to give both vectors to a third party, which will perform the non-linear part of the algorithm. For a vector of bits $(x_0,\dots,x_n) \in \{0,1\}^{n+1}$, the operation $\mathbf{any}_{j=0}^{n} x_j$ returns 1 iff there exists $j \in [0,\dots,n]$ such that $x_j = 1$. Algorithm 2 simply looks for the 0 coordinate in the two vectors. We have:

Algorithm 2 ComparisonFinal $((d_{0,0},\dots,d_{0,n}), (d_{1,0},\dots,d_{1,n}))$

```
1: Assign $b_0 := \mathbf{any}_{j=0}^{n}(d_{0,j} = 0)$
2: Assign $b_1 := \mathbf{any}_{j=0}^{n}(d_{1,j} = 0)$
3: return $b_0$ and $b_1$
```

In the below theorem, we again consider bit-decomposed integers $v_0 = \sum_{j=0}^{n-1} 2^{n-1-j} \cdot v_{0,j}$ and $v_1 = \sum_{j=0}^{n-1} 2^{n-1-j} \cdot v_{1,j}$; we view the bits $v_{i,j}$ as elements of $\{0,1\} \subset \mathbb{F}_q$. The following theorem is proven in the full version:

Theorem 4.4. *Suppose n is such that $2 + 4 \cdot (2^n - 1) < q$. Then for every $v_0, v_1 \in \mathbb{F}_q$*

$$(v_0 \leq v_1, v_1 \leq v_0) = \text{ComparisonFinal}(\text{ComparisonInitial}(\vec{v_0}, \vec{v_1})),$$

where $\vec{v_0}, \vec{v_1}$ are the bit-decomposition of v_0, v_1, respectively. Moreover, for every $i \in \{0,1\}$:

- *If $v_i \leq v_{1-i}$ then there exists exactly one $j \in \{0,\dots,n\}$ such that $d_{i,j} = 0$. Moreover, j is distributed uniformly in $\{0,\dots,n\}$, and each $d_{i,k}$ for $k \neq j$ is distributed uniformly in $\mathbb{F}_q \setminus \{0\}$.*
- *If $v_i > v_{1-i}$ then the vector $(d_{i,0},\dots,d_{i,n})$ is distributed uniformly in $\mathbb{F}_q^{n+1} \setminus \{0\}^{n+1}$.*

We emphasize that Algorithm 1 uses only $\mathbb{F}_q$-linear operations throughout. A number of our below protocols conduct Algorithm 1 "homomorphically"; that is, they execute the algorithm on elements of an $\mathbb{F}_q$-module M which is unequal to $\mathbb{F}_q$ itself. As a basic example, Algorithm 1 may be executed on bits $(v_{0,j})_{j=0}^{n-1}$ and $(v_{1,j})_{j=0}^{n-1}$ which are *committed*, provided that the commitment scheme is homomorphic (its message, randomness and commitment spaces should be $\mathbb{F}_q$-modules, and its commitment function an $\mathbb{F}_q$-module homomorphism). Furthermore, Algorithm 1 may be conducted on additive $\mathbb{F}_q$-shares of the bits $(v_{0,j})_{j=0}^{n-1}$ and $(v_{1,j})_{j=0}^{n-1}$.

In this latter setting, sense must be given to the affine additive constants ± 1. As in [28], we specify that these be shared in the obvious way; that is, we stipulate that P_0 and P_1 use the shares 0 and ± 1, respectively.

4.3 The Semi-Honest Protocol

For simplicity, we first describe a protocol that securely computes this functionality in the setting of three-party computation with an honest majority and a *semi-honest* adversary. We give a maliciously secure version in Protocol 4.3 and prove its security in the full version.

Theorem 4.5. *If Π_{CT} is a secure coin-tossing protocol, G is a pseudorandom generator, and the two clients communicate using symmetric authenticated encryption with pseudorandom ciphertexts, then Protocol 4.2 securely computes Functionality 4.1 in the presence of a semi-honest adversary corrupting at most one party. Each party sends/receives $O(n^2 + \lambda)$ bits, where n is the length of the input and λ is the sec. parameter.*

4.4 The Maliciously Secure Protocol

We now give our malicious protocol for Functionality 4.1 in Protocol 4.3. To ease notation, we denote $N = \{0,\dots,n-1\}$. We already gave an overview of the protocol as part of the introduction. The parties commit to the inputs, share their inputs, commit to the shares, and prove that all of those are consistent. Then, each party can operate on the shares it received (as in

PROTOCOL 4.2 (Semi-honest secure comparison protocol)**.**

- **Input:** P_0 and P_1 hold integers v_0 and v_1, respectively, in $\{0,\dots,2^n-1\}$. P^* has no input.
- **The protocol:**
  1. P_0 and P_1 engage in the coin-tossing procedure Π_{CT} (see full version for a formal definition) to obtain a λ-bit shared secret s.
  2. Each party P_i (for $i \in \{0,1\}$) computes the bit decomposition $v_i = \sum_{j=0}^{n-1} 2^{n-1-j} \cdot v_{i,j}$, for bits $v_{i,j} \in \{0,1\}$.
  3. For each $j \in \{0,\dots,n-1\}$, P_i computes a random additive secret-sharing $v_{i,j} = \langle v_{i,j}\rangle_0^q + \langle v_{i,j}\rangle_1^q$ in $\mathbb{F}_q$. P_i sends the shares $\left(\langle v_{i,j}\rangle_{1-i}^q\right)_{j=0}^{n-1}$ to P_{1-i}.
  4. After receiving the shares $\left(\langle v_{1-i,j}\rangle_i^q\right)_{j=0}^{n-1}$, from P_{i-1}, P_i executes Algorithm 1 on the appropriate shares; that is, it evaluates
  $$\left(\left(\langle d_{0,j}\rangle_i^q\right)_{j=0}^{n}, \left(\langle d_{1,j}\rangle_i^q\right)_{j=0}^{n}\right) \leftarrow \mathsf{ComparisonInitial}\left(\left(\langle v_{0,j}\rangle_i^q\right)_{j=0}^{n-1}, \left(\langle v_{1,j}\rangle_i^q\right)_{j=0}^{n-1}\right),$$
  where all internal random coins are obtained from $G(s)$.
  5. P_i sends the output shares $\left(\langle d_{0,j}\rangle_i^q\right)_{j=0}^{n}, \left(\langle d_{1,j}\rangle_i^q\right)_{j=0}^{n}$ to P^*.
  6. After receiving all shares, P^* reconstructs for every $j \in \{0,\dots,n\}$:
  $$d_{0,j} := \langle d_{0,j}\rangle_0^q + \langle d_{0,j}\rangle_1^q \quad \text{and} \quad d_{1,j} := \langle d_{1,j}\rangle_0^q + \langle d_{1,j}\rangle_1^q\ ,$$
  and finally executes Algorithm 2 to receive $(b_0,b_1) := \mathsf{ComparisonFinal}\left((d_{0,j})_{j=0}^{n}, (d_{1,j})_{j=0}^{n}\right)$.
  7. P^* sends b_i to P_i.
- **Output:** Each P_i outputs b_i. P^* outputs (b_0,b_1).

the semi-honest protocol), but also on the commitment that it holds on the other parties' share. When P^* receives the shares of d from party P_i, it also receives a commitment of what P_{1-i} is supposed to send. The server can therefore check for consistency, and that no party cheated. Moreover, each party P_i can also compute commitments of the two vectors $\vec{d}_0, \vec{d}_1$. When the server comes to prove P_i that $v_i \leq v_{1-i}$, it has to show that there is a 0 coordinate in its vector $\vec{d}_i$. This is possible using one-out-of-many proof. See the full version for the full security proof. We note that the functionality is slightly different than that of Functionality 4.1.

4.5 Bank-to-Client

We show in Appendix A a two-party (Bank-to-Client) version of the protocol, where the bank does not just facilitate the computation but also provides input. In a nutshell, the parties use linear homomorphic encryption – ElGamal encryption – instead of secret sharing.

5 Prime Match System Performance

We report benchmarks of Prime Match in two different environments, a Proof of Concept (POC) environment, and the production environment after refactoring the code to meet the requirements of the bank's systems. The former benchmarks can be used to value the performance of the comparison protocol for other applications in different systems.

Secure Minimum Protocol Performance: For the purposes of practical convenience, adoption, and portability, our client module is entirely browser-based, written in JavaScript. Its cryptographically intensive components are written in the C language with side-channel resistance, compiled using Emscripten into WebAssembly (which also runs natively in the browser). Our server is written in Python, and also executes its cryptographically intensive code in C. Both components are multi-threaded—using WebWorkers on the client side and a thread pool on the server's—and can execute arbitrarily many concurrent instances of the protocol in parallel (i.e., constrained only by hardware). All players communicate by sending binary data on WebSockets (all commitments, proofs, and messages are serialized).

We run our experiments on commodity hardware throughout since our implementation is targeted to a real-world application where clients hold conventional computers. In particular, one of the two clients runs on an Intel Core i7 processor, with 6 cores, each 2.6GHz, and another one runs on an Intel Core i5, with 4 cores, each 2.00 GHz. Both of them are Windows machines. Our server runs in a Linux AWS instance of type c5a.8xlarge, with 32 vCPUs. In the first scenario, we run the client-to-bank inventory matching protocol for two clients where we process each client one by one against the bank's inventory. In Table 2 we report the performance for a different number of registered orders per client $(100, 200, \dots, 10000)$. *Latency* refers to the total time it takes to process all the orders from both clients (in seconds). *Throughput* measures the num-

PROTOCOL 4.3 (Maliciously secure comparison protocol Π_{comp}).
Input: P_0 and P_1 hold integers v_0 and v_1, respectively, in $\{0,\dots,2^n-1\}$. P^* has no input.
Setup phase: A coin-tossing protocol Π_{CT}, and a commitment scheme $(\mathsf{Gen},\mathsf{Com})$, are chosen (see Sect. 2).
The protocol:

1. Commit $V_i \leftarrow \mathsf{Com}(v_i; r_i)$, and send V_i to P^*. P^* delivers V_i to P_{1-i}.
2. Engage with P_{1-i} in the coin-tossing procedure Π_{CT}, and obtain a λ-bit shared s.
3. Compute the bit decomposition $v_i = \sum_{j\in N} 2^{n-1-j}\cdot v_{i,j}$, for bits $v_{i,j}\in\{0,1\}$.
4. For each $j \in N$:
   (a) Compute a random additive secret-sharing $v_{i,j} = \langle v_{i,j}\rangle_0^q + \langle v_{i,j}\rangle_1^q$ in $\mathbb{F}_q$.
   (b) Commit $V_{i,j,k} \leftarrow \mathsf{Com}(\langle v_{i,j}\rangle_k^q;\ r_{i,j,k})$ for each $k\in\{0,1\}$.
   (c) Open $V_{i,j,1-i}$ by sending $\langle v_{i,j}\rangle_{1-i}^q$ and $r_{i,j,1-i}$ to P_{1-i}.
5. Send the full array $\left(V_{i,j,k}\right)_{j,k=0}^{n-1,1}$ to P_{1-i}.
6. Compute: $\pi_i \leftarrow \mathsf{ComEq.Prove}\left(V_i, \prod_{j=0}^{n-1}\left(V_{i,j,0}\cdot V_{i,j,1}\right)^{2^{n-1-j}}\right)$, $\pi_{i,j} \leftarrow \mathsf{BitProof.Prove}\left(V_{i,j,0}\cdot V_{i,j,1}\right)$, for all $j\in N$, P_i sends π_i, and $(\pi_{i,j})_{j=0}^{n-1}$ to P_{1-i}.
7. P_i receives π_{1-i}, and $(\pi_{1-i,j})_{j=0}^{n-1}$ and verifies the following:
   (a) The openings $\langle v_{1-i,j}\rangle_i^q$ and $r_{1-i,j,i}$ indeed open $V_{1-i,j,i}$, for $j\in N$,
   (b) $\mathsf{ComEq.Verify}\left(\pi_{1-i}, V_{1-i}, \prod_{j=0}^{n-1}\left(V_{1-i,j,0}\cdot V_{1-i,j,1}\right)^{2^{n-1-j}}\right)$,
   (c) $\mathsf{BitProof.Verify}\left(\pi_{1-i,j}, V_{1-i,j,0}\cdot V_{1-i,j,1}\right)$ for each $j\in N$.
   If any of these checks fail, P_i aborts.
8. P_i runs Algorithm 1, in parallel, on the shares $\langle v_{k,j}\rangle_i^q$, the randomnesses $r_{k,j,i}$, and the commitments to the *other party*'s shares $V_{k,j,1-i}$ (all for $j\in N$ and $k\in\{0,1\}$). That is, P_i runs:

$$\left(\left(\langle d_{0,j}\rangle_i^q\right)_{j=0}^n, \left(\langle d_{1,j}\rangle_i^q\right)_{j=0}^n\right) \leftarrow \mathsf{ComparisonInitial}\left(\left(\langle v_{0,j}\rangle_i^q\right)_{j=0}^{n-1}, \left(\langle v_{1,j}\rangle_i^q\right)_{j=0}^{n-1}\right),$$
$$\left(\left(s_{0,j,i}\right)_{j=0}^n, \left(s_{1,j,i}\right)_{j=0}^n\right) \leftarrow \mathsf{ComparisonInitial}\left(\left(r_{0,j,i}\right)_{j=0}^{n-1}, \left(r_{1,j,i}\right)_{j=0}^{n-1}\right),$$
$$\left(\left(D_{0,j,1-i}\right)_{j=0}^n, \left(D_{1,j,1-i}\right)_{j=0}^n\right) \leftarrow \mathsf{ComparisonInitial}\left(\left(V_{0,j,1-i}\right)_{j=0}^{n-1}, \left(V_{1,j,1-i}\right)_{j=0}^{n-1}\right),$$

   using the same shared randomness s for all internal coin flips.
9. P_i sends $\left(\langle d_{0,j}\rangle_i^q\right)_{j=0}^n, \left(\langle d_{1,j}\rangle_i^q\right)_{j=0}^n$ and the randomnesses $\left(s_{0,j,i}\right)_{j=0}^n, \left(s_{1,j,i}\right)_{j=0}^n$ to P^*.

Party P^* (Output reconstruction): After receiving all shares, P^* proceeds as follows:

1. Reconstruct for each $j\in N$:

$$d_{0,j} := \langle d_{0,j}\rangle_0^q + \langle d_{0,j}\rangle_1^q, \qquad s_{0,j} := s_{0,j,0} + s_{0,j,1}$$
$$d_{1,j} := \langle d_{1,j}\rangle_0^q + \langle d_{1,j}\rangle_1^q, \qquad s_{1,j} := s_{1,j,0} + s_{1,j,1}$$

2. Finally, P^* executes Algorithm 2, that is: $(b_0,b_1) := \mathsf{ComparisonFinal}\left(\left(d_{0,j}\right)_{j=0}^n, \left(d_{1,j}\right)_{j=0}^n\right)$
3. For each $i\in\{0,1\}$, if b_i is true, then P^* re-commits $D_{i,j} := \mathsf{Com}(d_{i,j}; s_{i,j})$ for each $j\in N$, computes $\pi'_i \leftarrow \mathsf{OneMany.Prove}\left(\left(D_{i,j}\right)_{j=0}^n\right)$, and finally sends π'_i to P_i. Otherwise, P^* sends $\perp$ to P_i. P^* outputs both b_0 and b_1.

Each Party P_i (output reconstruction):

1. If receives a proof π'_i, compute $D_{i,j} := \mathsf{Com}(\langle d_{i,j}\rangle_i^q; s_{i,j,i})\cdot D_{i,j,1-i}$, for each $j\in N$, and then verifies $\mathsf{OneMany.Verify}\left(\pi'_i, \left(D_{i,j}\right)_{j=0}^n\right)$. If verification passes then output true, otherwise, output false.

	Number of Symbols	Latency (sec)	Throughput (transactions/ sec)	Matching received msg Size (MB)	Matching sent msg Size (MB)
Bank-to-client	100	9.903 (±0.174)	10.09	0.215	0.521
	200	19.533 (±0.530)	10.23	0.430	1.040
	500	46.223(±0.787)	10.81	1.076	2.597
	1000	95.396 (±2.063)	10.48	2.152	5.194
	2000	183.186 (±2.512)	10.91	4.304	10.382
	4000	356.740 (±2.149)	11.21	8.608	20.762
	10000	941.813 (±18.465)	10.61	21.520	51.902
Client-to-client	100	11.15 (±0.060)	8.96	0.972	1.549
	200	20.636 (±0.525)	9.69	1.945	3.096
	500	51.493(±2.343)	9.71	4.863	7.737
	1000	101.051 (±2.587)	9.89	9.727	15.472
	2000	208.813 (±1.479)	9.57	19.454	30.942
	4000	390.510 (±3.020)	10.24	38.908	61.882
	10000	1064.443 (±70.439)	9.39	97.270	154.702

Table 2: Performance of Bank-to-Client matching for two clients, and Client-to-Client matching.

Number of Symbols	Latency (sec)	Throughput (transactions/ sec)	Registration received msg Size (MB)	Registration sent msg Size (MB)	Matching received msg Size (MB)	Matching sent msg Size (MB)
100	23.85 (±9.19)	8.38	0.062	0.036	16.225	4.510
200	29.49 (±2.85)	13.56	0.124	0.073	32.451	9.021
500	65.04 (±1.41)	15.37	0.310	0.184	81.128	22.552
1000	136.93 (±3.32)	14.60	0.621	0.370	162.257	45.105
2000	244.37 (±10.31)	16.36	1.243	0.749	324.507	90.210
3000	428.48 (±32.19)	14.00	1.865	1.128	486.739	135.260
4000	1077.60 (±27.46)	7.42	2.487	1.507	647.758	180.043
5000	1314.44 (±39.15)	7.60	3.109	1.886	809.191	224.641
6000	1558.58 (±46.96)	7.69	3.731	2.265	971.875	269.997

Table 3: Performance of Bank-to-Client Prime Match for two clients. The number of symbols processed in the production code is double since clients provide both a minimum and a maximum quantity to be matched. This is reflected at the Throughput column which is multiplied by a factor of two.

ber of transactions per second. The number of orders/symbols processed per second is approximately 10. We also report the message size (in MB) for the message sent from each client to the server during the registration phase and the matching phase. We further record the size of the messages received from the server to each client. The bandwidth is 300 Mbps.

In the second scenario, we run the client-to-client inventory matching protocol (the comparison protocol) for two clients where we try to match the orders of the two clients via the bank. In this case we assume that the bank has no inventory. We report the performance in Table 2. The number of orders/symbols processed per second is approximately 10 in this case too.

Prime Match Performance in Production: Figure 5 shows a sketch of both the bank's network tiering and the application architecture of Prime Match. The right side of the diagram shows that the clients are able to access the Prime Match UI through the bank's Markets portal with the correct entitlement.

The application UI for Prime Match is hosted on the bank's internal cloud platform. After each client is getting authenticated by the bank's Markets portal, it can access the application on the browser which establishes web socket connections to the server. The server is hosted on the bank's trade management platform.

The bank's network tiering exists to designate the network topology in and between approved security gateways (firewalls). The network traffic from the client application is handled by tier 1 which helps the bank's Internet customers achieve low latency, secured, and accelerated content access to internally hosted applications. Then, the traffic will be subsequently forwarded to a web socket tunnel in tier 2 and gets further directed to the server in tier 3.

During the axe registration phase, the client logins to Prime Match from his/her web browser and uploads a file of orders (symbol, directions, quantity) which are encrypted on the browser based on our MPC protocol. The client is uploading a minimum (threshold) and a maximum (full) quantity per symbol. Prime Match first processes the encrypted threshold quantities of all clients and then it processes the full quantities. This process implements the range functionality of Appendix B. Note that for the partial matches, the full quantity is never revealed to the server. Moreover, if there is no match no quantity is revealed to the server. Figure 6 shows a complete run of an auction after the matching phase where the final matched quantities of some test symbols/securities with the corresponding fixed spread are decided.

Our protocol runs in production every 30 minutes. There are two match runs each hour and matching starts at xx:10 and xx:40. Axe registration starts 8 minutes before matching (xx:02 and xx:32). The matching process finishes at various times (as shown in the experiments section) based on the number of symbols. Based on the application requirements (up to 5000 symbols in the US and 10-60 clients) it does not finish after xx:55.

For running the code in production in the bank's environment, the two clients run on the same type of Windows machines whose specs are Intel XEON CPU E3-1585L, with 4 cores, each 3.00 GHz. The server runs on the bank's trade management platform with 32 vCPUs. In Table 3 we report the performance. Moreover, we discuss challenges we faced during the implementation in Appendix C.

6 Acknowledgements

We would like to thank Mike Reich, Vaibhav Popat, Sitaraman Rajamani, Dan Stora, James Mcilveen, Oluwatoyin Aguiyi, Niall Campbell, Wanyi Jiang, Grant McKenzie, Steven Price, Vinay Gayakwad, Srikanth Veluvolu, Noel Peters for their great efforts and help to move Prime Match in production. Last but not least, we would like to thank our executive sponsor Jason Sippel. The paper is based on joint patents [3, 17]

References

[1] Amit Agarwal, Stanislav Peceny, Mariana Raykova, Phillipp Schoppmann, and Karn Seth. Communication efficient secure logistic regression. Cryptology ePrint Archive, Paper 2022/866, 2022. https://eprint.iacr.org/2022/866.

[2] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 805–817. ACM, 2016.

[3] Gilad Asharov, Tucker Balch, Hans Buehler, Richard Hua, Antigoni Polychroniadou, and Manuela Veloso. Systems and methods for privacy-preserving inventory matching, filed Dec. 6, 2019.

[4] Gilad Asharov, Tucker Hybinette Balch, Antigoni Polychroniadou, and Manuela Veloso. PPDPs: Privacy-preserving dark pools. In *19th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2020)*, 2020. Extended abstract.

[5] Samiran Bag, Feng Hao, Siamak F Shahandashti, and Indranil Ghosh Ray. Seal: Sealed-bid auction without auctioneers. *IEEE Transactions on Information Forensics and Security*, 15:2042–2052, 2019.

[6] Tucker Balch, Benjamin E. Diamond, and Antigoni Polychroniadou. Secretmatch: Inventory matching from fully homomorphic encryption. In *Proceedings of the First ACM International Conference on AI in Finance*, ICAIF '20, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Boaz Barak, Ran Canetti, Yehuda Lindell, Rafael Pass, and Tal Rabin. Secure computation without authentication. *J. Cryptol.*, 24(4):720–760, 2011.

[8] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *Financial Cryptography and Data Security, FC 2009*, volume 5628, pages 325–343. Springer, 2009.

[9] John Cartlidge, Nigel P. Smart, and Younes Talibi Alaoui. Mpc joins the dark side. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 148–159. Association for Computing Machinery, 2019. Full version.

[10] John Cartlidge, Nigel P. Smart, and Younes Talibi Alaoui. Multi-party computation mechanism for anonymous equity block trading: A secure implementation of turquoise plato uncross. Cryptology ePrint Archive, Report 2020/662, 2020. https://ia.cr/2020/662.

[11] Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In *SCN*, 2010.

[12] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *Advances in Cryptology - CRYPTO 2018*, volume 10993, pages 34–64. Springer, 2018.

[13] Mariana Botelho da Gama, John Cartlidge, Antigoni Polychroniadou, Nigel P. Smart, and Younes Talibi Alaoui. Kicking-the-bucket: Fast privacy-preserving trading using buckets. Financial Cryptography, 2022.

[14] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography, TCC 2006*, volume 3876, pages 285–304. Springer, 2006.

[15] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Homomorphic encryption and secure comparison. *Int. J. Appl. Cryptogr.*, 1(1):22–31, 2008.

[16] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. A correction to 'efficient and secure comparison for on-line auctions'. *Int. J. Appl. Cryptogr.*, 1(4):323–324, 2009.

[17] Benjamin Diamond and Antigoni Polychroniadou. Privacy-preserving inventory matching with security against malicious adversaries, filed Jan. 28, 2021.

[18] Marc Fischlin. A cost-effective pay-per-multiplication comparison method for millionaires. In *CT-RSA 2001*, volume 2020, pages 457–472. Springer, 2001.

[19] Hisham S Galal and Amr M Youssef. Publicly verifiable and secrecy preserving periodic auctions. In *International Conference on Financial Cryptography and Data Security*, pages 348–363. Springer, 2021.

[20] Jens Groth and Markulf Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In *Advances in Cryptology – EUROCRYPT 2015*, volume 9057 of *Lecture Notes in Computer Science*, pages 253–280. Springer Berlin Heidelberg, 2015.

[21] Jonathan Katz, Samuel Ranellucci, Mike Rosulek, and Xiao Wang. Optimizing authenticated garbling for faster secure two-party computation. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 365–391. Springer, 2018.

[22] Hsiao-Ying Lin and Wen-Guey Tzeng. An efficient solution to the millionaires' problem based on homomorphic encryption. In *Applied Cryptography and Network Security, Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, Proceedings*, volume 3531 of *Lecture Notes in Computer Science*, pages 456–466, 2005.

[23] Helger Lipmaa and Tomas Toft. Secure equality and greater-than tests with sublinear online complexity. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 645–656. Springer, 2013.

[24] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient comparison for secure multi-party computation. In *Financial Cryptography and Data Security - FC 2021*, volume 12674, pages 249–270. Springer, 2021.

[25] Fabio Massacci, Chan Nam Ngo, Jing Nie, Daniele Venturi, and Julian Williams. Futuresmex: Secure, distributed futures market exchange. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 335–353. IEEE Computer Society, 2018.

[26] Chan Nam Ngo, Fabio Massacci, Florian Kerschbaum, and Julian Williams. Practical witness-key-agreement for blockchain-based dark pools financial trading.

[27] Takashi Nishide and Kazuo Ohta. Constant-round multiparty computation for interval test, equality test, and comparison. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 90-A(5):960–968, 2007.

[28] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. In *Proceedings on Privacy Enhancing Technologies*, volume 2019, pages 26–49, 2019.

[29] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 21–37. ACM, 2017.

A Bank-to-Client Matching

In this section, we present a two-party variant of our main three-party comparison protocol which can be used for bank-to-client matching. We describe the functionality and then the protocol.

FUNCTIONALITY A.1 (Two-party min, $\mathcal{F}_{\min}^{\mathsf{B2C}}$.).
Input: The server P_0 holds v_0, and P_1 holds v_1, both in $\{0,\ldots,2^n-1\}$.
Output: Both parties receive $\min\{v_0,v_1\}$.

PROTOCOL A.2 (Two-party protocol $\Pi_{\min}^{\mathsf{B2C}}$).
Input: P_0 and P_1 hold integers v_0 and v_1, respectively, in $\{0,\ldots,2^n-1\}$.
Setup phase: A commitment scheme (Com) and an encryption scheme $(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec})$ are chosen. P_0 runs $(pk,sk)\leftarrow\mathsf{Gen}(1^\lambda)$, and authenticates its public key pk with P_1.

The protocol:

1. **P_0 proceeds as follows: (Encryption):**
   (a) Commit $V_0\leftarrow\mathsf{Com}(v_0;r_0)$, and send V_0 to P_1.
   (b) Compute the bit decomposition $v_0=\sum_{j\in N}2^{n-1-j}\cdot v_{0,j}$, for bits $v_{0,j}\in\{0,1\}$.
   (c) For each $j\in N$: compute additive homomorphic encryptions $A_j=\mathsf{Enc}_{pk}(v_{0,j})$.
   (d) Send the full array $(A_j)_{j=0}^{N-1}$ to P_1.
   (e) Compute:
      i. $\pi\leftarrow\mathsf{ComEq.Prove}\left(V_0,\prod_{j=0}^{N-1}(A_j)^{2^j}\right)$,
      ii. $\pi_j\leftarrow\mathsf{BitProof.Prove}(A_j)$, for all $j\in N$,
   P_0 sends π, and $(\pi_j)_{j=0}^{N-1}$ to P_1.
2. **P_1 proceeds as follows: (Computing the minimum):**
   Receive $(\pi,(\pi_j)_{j=0}^{N-1})$ and verify the following:
   (a) $\mathsf{ComEq.Verify}\left(\pi,V_0,\prod_{j=0}^{N-1}(A_j)^{2^j}\right)$,
   (b) $\mathsf{BitProof.Verify}(\pi_j,A_j)$ for each $j\in N$. If any of these checks fail, P_1 aborts.
   (c) Run Algorithm 1, in parallel, on the ciphertexts A_j and on its own secret inputs. That is, P_1 runs:
   $$\left((D_{0,j})_{j=0}^{N},(D_{1,j})_{j=0}^{N}\right)\leftarrow$$
   $$\mathsf{ComparisonInitial}\left((A_j)_{j=0}^{N-1},(v_{1,j})_{j=0}^{N-1}\right)$$
   recall that the algorithm shuffles the two result vectors inside.
   (d) P_1 sends $\left((D_{0,j})_{j=0}^{N},(D_{1,j})_{j=0}^{N}\right)$ to P_0.
3. **Party P_0 (Output reconstruction):**
   (a) Decrypt $d_{i,j}=\mathsf{dec}_{sk}(D_{i,j})$ for each $i=\{0,1\}$ and $j=\{0,\ldots,n-1\}$
   (b) Execute Algorithm 2, that is:
   $$(b_0,b_1):=\mathsf{ComparisonFinal}\left((d_{0,j})_{j=0}^{N},(d_{1,j})_{j=0}^{N}\right).$$
   (c) Write u for the index such that b_u is true and sets $u:=0$ if both are.
   (d) Compute $\pi'\leftarrow\mathsf{OneMany.Prove}\left((D_{u,j})_{j=0}^{n-1}\right)$, and sends π' and u to P_1. If $u=1$ then send also v_1.
4. **Party P_1 (output reconstruction):**
   (a) P_1 verifies $\mathsf{OneMany.Verify}\left(\pi',(D_{u,j})_{j=0}^{N-1}\right)$. If verification passes, then P_1 outputs v_u.

Theorem A.3. *Protocol A.2 securely computes Functionality A.1 in the presence of a malicious P_0 or a semi-honest P_1, assuming secure commitments and zero-knowledge functionalities.*

Concrete instantiation. We implement Protocol A.2 using the ElGammal encryption scheme, where the message m is part of the exponent (thereby we get additive homomorphism). I.e., for a public key $h\in\mathbb{G}$, $\mathsf{Enc}_pk(m)=(g^r,h^rg^m)$. Note that P_1 does not have to decrypt the ciphertexts, but just identify an encryption of 0. Given a secret key x such that $h=g^x$ and a ciphertext $c=(c_1,c_2)$, this is done by simply comparing c_2/c_1^x to g^0.

Implementing Functionality 3.1. To implement the bank-to-client functionality, the parties invoke $2|U|$ times the two-party minimum functionality. For each possible symbol in U, the parties invoke the minimum functionality where once the bank inputs its interest in symbols to buy and the client in sell, and one time where the bank inputs whether it wishes to sell and the client inputs whether it wishes to buy. If a party is not interested in some symbol or in a particular side, it simply inputs 0. The results of all minimum invocations will reveal the matches. Security follows from composition. We have:

Corollary A.4. *The above protocol securely computes Functionality 3.1 ($\mathcal{F}_{\mathsf{B2C}}$) in the presence of a semi-honest bank or a malicious client.*

B Range Bank-to-Client Functionality

In this section, we consider the setting in which the client submits a minimum and a maximum amount per stock instead of a single amount. The current system is set to accept a minimum and a maximum quantity from the clients to be matched against the bank's inventory. As a first step, the bank runs Functionality 3.1 on every client one by one only on their minimum quantity until the bank exhausts its inventory. For the case where the bank did not exhaust its inventory after processing all the clients, it runs again Functionality 3.1 on every client one by one on their maximum quantity. This process is presented in Functionality B.1. The requirement on the minimum and maximum quantity is imposed by the business for this use case as a greedy approach in an effort to satisfy more clients with the bank's inventory. This is not a necessary requirement, but the bank views it as an additional feature to satisfy more clients. This greedy approach is not ideal since, for instance, the minimum quantity of the first client can exhaust the full inventory of the bank. We leave it as an open problem to devise a secure optimization algorithm for better allocations across all clients.

This additional range feature is considered only when we seek for matches between the bank and a client in an effort to maximize the number of matches the bank can accommodate from its own inventory. Thart said, we do not explicitly consider it for the client-to-client matching.

FUNCTIONALITY B.1 ($\mathcal{F}_{\mathrm{B2C}}$–Range Bank-to-client functionality)**.**
The functionality is parameterized by the set of all possible securities to trade, a set U.
Input: The bank P^* inputs lists of orders $(\mathsf{symb}_i^*, \mathsf{side}_i^*, \mathsf{amount}_i^*)$ where $\mathsf{symb}_i \subseteq U$ is the security, $\mathsf{side}_i^* \in \{\mathtt{buy}, \mathtt{sell}\}$ and amount_i^* is an integer. A client P_i sends its list of the same format but with a minimum and a maximum amount, $(\mathsf{symb}_i^C, \mathsf{side}_i^C, \mathsf{MinAmount}_i^C, \mathsf{MaxAmount}_i^C)$.
Output: Initialize a list of Matches M. Choose a random order O over all clients.

- For the next client j in O and for every i, such that $\mathsf{symb}_i^* = \mathsf{symb}_j^C$ and $\mathsf{side}_i^* \neq \mathsf{side}_j^C$, add $(\mathsf{symb}_i^*, \mathsf{side}_i^*, \mathsf{side}_j^C, v = \min\{\mathsf{amount}_i^*, \mathsf{MinAmount}_j^C\})$ to M and update $\mathsf{amount}_i^* = \mathsf{amount}_i^* - v$.

Then, check maximum amounts for the j in M:

- For the next j in O and every i such that $\mathsf{symb}_i^* = \mathsf{symb}_j^C$ and $\mathsf{side}_i^* \neq \mathsf{side}_j^C$, add $(\mathsf{symb}_i^*, \mathsf{side}_i^*, \mathsf{side}_j^C, v = \min\{\mathsf{amount}_i^*, \mathsf{MaxAmount}_j^C\})$ to M and update $\mathsf{amount}_i^* = \mathsf{amount}_i^* - v$.

Implementing this protocol is quite straightforward, given the protocols we already have. The bank simply runs the minimum functionality against (random) ordering of the clients with their minimum amount, and then with their maximum amount, while updating the amount of each symbol along the way.

C Challenges in Implementation

Privacy-preserving auctions have been the holy grail of practical MPC since [8]. However, no financial institutions took a step forward to use such tools. To begin with, we got connected to the bank's business department of quantitative research to test the appetite for a form of privacy-preserving

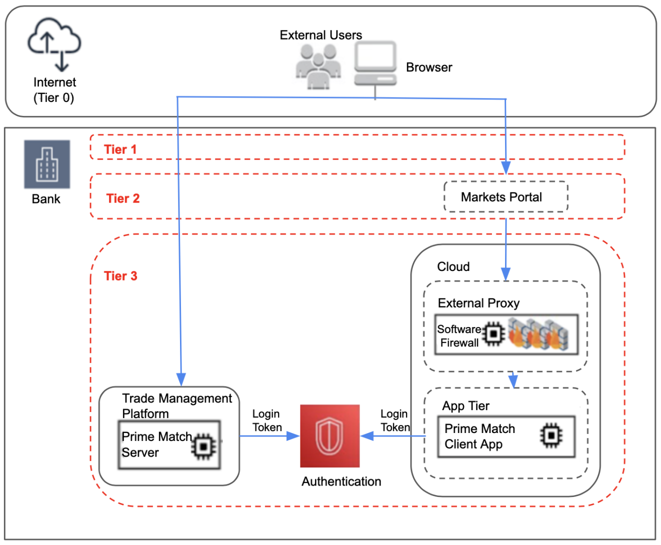


Figure 5: Prime Match system architecture connecting external users/clients to the bank's network.

auctions. Thus, the Axe inventory use case was chosen.

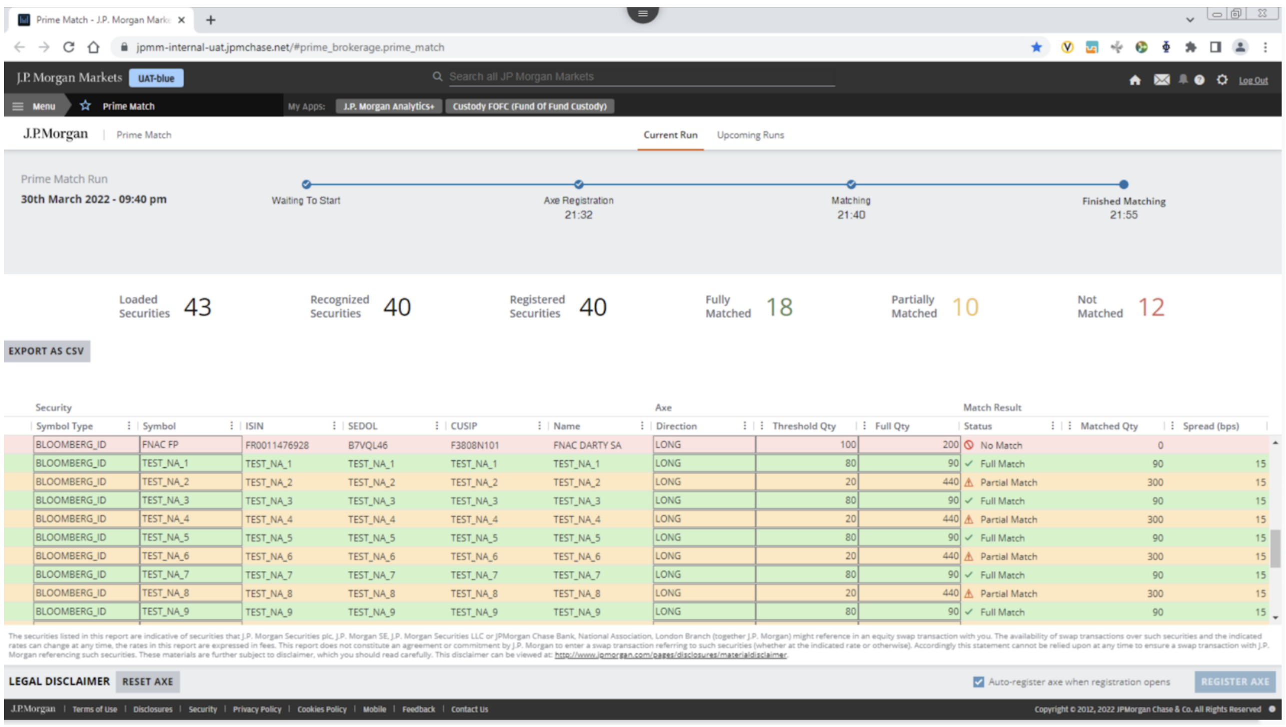
Pre-Production Challenges. Releasing a secure multiparty computation (MPC) product in a big organization that had no other products utilizing these techniques was itself challenging. Furthermore, there were no other products across the street (market) to showcase the feasibility of such a solution.

As a first step, a proof of concept (POC) was implemented to show its feasibility.

Given the innovative nature of the product, the green light for production was given after a long process. More specifically, the organization decided to test the appetite for the possible product of its valuable clients (hedge funds). Almost 20 hedge funds were visited in 6 months to demo the POC and to hear their highly appreciated feedback. Given the client feedback, the organization decided to move another step forward and ask the clients under what conditions they will utilize such a product. In particular, the clients provided a set of different features and requirements they would like to see in the product. Notable requirements were:

- No communication with other clients, only communication with the bank.
- No resources for preprocessing data.
- No installation of code on client's machines - a web-based application was required.
- Stronger security guarantees than semi-honest security.
- Peer review of the solution.

The POC was updated to satisfy the above requirements. Another round of demos to clients was conducted. After several months the business gave the green light to build the product and allocated resources to enable it. The team consisted of



Prime Match Run
30th March 2022 - 09:40 pm

Waiting To Start — Axe Registration 21:32 — Matching 21:40 — Finished Matching 21:55

Loaded Securities 43 | Recognized Securities 40 | Registered Securities 40 | Fully Matched 18 | Partially Matched 10 | Not Matched 12

Security						Axe			Match Result		
Symbol Type	Symbol	ISIN	SEDOL	CUSIP	Name	Direction	Threshold Qty	Full Qty	Status	Matched Qty	Spread (bps)
BLOOMBERG_ID	FNAC FP	FR0011476928	B7VQL46	F3808N101	FNAC DARTY SA	LONG	100	200	No Match	0	
BLOOMBERG_ID	TEST_NA_1	TEST_NA_1	TEST_NA_1	TEST_NA_1	TEST_NA_1	LONG	80	90	✓ Full Match	90	15
BLOOMBERG_ID	TEST_NA_2	TEST_NA_2	TEST_NA_2	TEST_NA_2	TEST_NA_2	LONG	20	440	⚠ Partial Match	300	15
BLOOMBERG_ID	TEST_NA_3	TEST_NA_3	TEST_NA_3	TEST_NA_3	TEST_NA_3	LONG	80	90	✓ Full Match	90	15
BLOOMBERG_ID	TEST_NA_4	TEST_NA_4	TEST_NA_4	TEST_NA_4	TEST_NA_4	LONG	20	440	⚠ Partial Match	300	15
BLOOMBERG_ID	TEST_NA_5	TEST_NA_5	TEST_NA_5	TEST_NA_5	TEST_NA_5	LONG	80	90	✓ Full Match	90	15
BLOOMBERG_ID	TEST_NA_6	TEST_NA_6	TEST_NA_6	TEST_NA_6	TEST_NA_6	LONG	20	440	⚠ Partial Match	300	15
BLOOMBERG_ID	TEST_NA_7	TEST_NA_7	TEST_NA_7	TEST_NA_7	TEST_NA_7	LONG	80	90	✓ Full Match	90	15
BLOOMBERG_ID	TEST_NA_8	TEST_NA_8	TEST_NA_8	TEST_NA_8	TEST_NA_8	LONG	20	440	⚠ Partial Match	300	15
BLOOMBERG_ID	TEST_NA_9	TEST_NA_9	TEST_NA_9	TEST_NA_9	TEST_NA_9	LONG	80	90	✓ Full Match	90	15

The securities listed in this report are indicative of securities that J.P. Morgan Securities plc, J.P. Morgan SE, J.P. Morgan Securities LLC or JPMorgan Chase Bank, National Association, London Branch (together J.P. Morgan) might reference in an equity swap transaction with you. The availability of swap transactions over such securities and the indicated rates can change at any time, the rates in this report are expressed in fees. This report does not constitute an agreement or commitment by J.P. Morgan to enter a swap transaction referring to such securities (whether at the indicated rate or otherwise). Accordingly this statement cannot be relied upon at any time to ensure a swap transaction with J.P. Morgan referencing such securities. These materials are further subject to disclaimer, which you should read carefully. This disclaimer can be viewed at: http://www.jpmorgan.com/pages/disclosures/materialdisclaimer.

✓ Auto-register axe when registration opens

Copyright © 2012, 2022 JPMorgan Chase & Co. All Rights Reserved

Prime Match - J.P. Morgan Markets
jpmm-internal-uat.jpmchase.net/#prime_brokerage.prime_match
J.P. Morgan Markets
UAT-blue
Search all JP Morgan Markets
Log Out
Menu
Prime Match
My Apps:
J.P. Morgan Analytics+
Custody FOFC (Fund Of Fund Custody)
J.P.Morgan
Prime Match
Current Run
Upcoming Runs
EXPORT AS CSV
LEGAL DISCLAIMER
RESET AXE
REGISTER AXE
J.P.Morgan | Terms of Use | Disclosures | Security | Privacy Policy | Cookies Policy | Mobile | Feedback | Contact Us

Figure 6: Prime Match User Interface of a completed matching run.

cryptographers, quants, tech quants, expert programmers on different topics (such as secure multiparty computation, UI, web-based development etc.), cybersecurity experts, product managers, legal, and stakeholders.

Production Challenges - Lessons Learned During the development of Prime Match, we faced several critical technical challenges. Those challenges were mainly due to the complex design of the organization's infrastructure for security measures.

- Coding in the Trade Management platform and the Markets Portal, see Figure 5, needed to follow certain practices (which easy maintenance too) which were not followed in the POC stage. POC was reprogrammed to follow current practices.
- Only a limited set of open-source libraries are allowed in the Trade Management platform. For example, importing the open-source library emscripten docker image to the Trade Management platform. This process went through rigorous request and approval processes to show that this library does not contain any code that would secretly send data to the internet.
- Integration to the markets portal which is an existing service for external clients, in order to access some of the organization's trading applications, was not easy since no other application required interaction between the server and the clients. Prior apps involve simple downloads of data from clients.
- Set up a pathway that allows network traffic to flow from the clients to the Prime Match server with the use of web socket protocol. In particular, setting up Psaas service in Tier 1 and 2, which stands for Perimeter Security as a Service. Extensive review and approval process were required.
- Placing a physical server and setting up an Apache WS tunnel in tier 1 to enable external clients to transact and transact within tight latency requirements. In particular, we had to implement the entire WS tunnel proxy from scratch since the Web Sockets protocol was not used within these platforms before.
- Run several internal tests ranging from UI requirements to cyber security requirements (such as timing attacks).
- Extensive code review from internal and external experts.
- Integrate the UI component with the markets portal which required extensive approvals.
- Integration of our code with WebAssembly to achieve near-native performance. To this end, we also had to handle garbage collection. See more details on the chosen programming languages in Section 5.

Overall each step required extensive review and approval processes. An important aspect that was not listed above is the last process of extensive testing, verification of the code, and beta testing the product with the help of a limited number of clients.