Prime Match: A Privacy-Preserving Inventory Matching System

Antigoni Polychroniadou, *J.P. Morgan;* Gilad Asharov, *Bar-Ilan University;*
Benjamin Diamond, Tucker Balch, Hans Buehler, Richard Hua,
Suwen Gu, Greg Gimler, and Manuela Veloso, *J.P. Morgan*

https://www.usenix.org/conference/usenixsecurity23/presentation/polychroniadou

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Prime Match: A Privacy-Preserving Inventory Matching System

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

J.P. Morgan

A Artifact Appendix

A.1 Abstract

In this work, we introduce secure multiparty computation in financial services by presenting a solution, called *Prime Match*, for matching orders in a stock exchange while maintaining the privacy of the orders. Information is revealed only if there is a match. Our central tool is a new protocol for secure comparison with malicious security, which can be of independent interest. In this artifact, we showcase the major claims of our paper titled "Prime Match: A Privacy-Preserving Inventory Matching System".

A.2 Description & Requirements

Prime Match involves a server/bank and (at least) a client which submits orders to buy or sell a particular stock/symbol, along with the intended quantity (number of shares), to the bank. 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 in the bank 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.

A.2.1 Security, privacy, and ethical concerns

No concerns.

A.2.2 Hardware dependencies

Our code can run on any commodity hardware since our implementation is targeted to a real-world application where clients hold conventional computers. For example, for our experiments, 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. However, it is possible for a reviewer to test our system by running both the server and client(s) on the same machine. If two clients are selected for a test on the same machine, one of the two clients needs to be executed in incognito mode from the browser.

[1]Currently at Bar-Ilan University, based on work that was conducted while at J.P. Morgan.

A.2.3 Software dependencies

For the purposes of practical convenience, adoption, and portability, our client module is entirely browser-based and written in JavaScript. Its cryptographically intensive components are written in the C language and 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 multithreaded—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. Our code is independent of the operating system (MacOS is recommended) and can run on any browser (Google Chrome is recommended).

A.2.4 Benchmarks

None, our code generates random inputs on the fly.

A.3 Set-up

A.3.1 Installation

Our library consists of multiple components. Its client is written in JavaScript, while we provide both Python and JavaScript implementations of the server. Both the Python server and the JavaScript client use C code, which is separated into its own folder. Next, we provide the installation guide to install both the server and the client (the same information is also provided in the Readme files of the repo).

Python Server Instructions:

Prerequisites. Install Python 3.8 and pip. Add ./src to the PYTHONPATH environment variable. Run pip3 install −r requirements.txt.

Installation. In the python folder directory execute:

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_C_FLAGS_
RELEASE="-DNDEBUG -O3" -DCMAKE_SHARED_LINKER_
FLAGS_RELEASE="" ..
make
```

JavaScript Client Instructions:

Prerequisites. Install Yarn (tested with version v1.22.4). To build the C components, download and install emscripten.

Installation. After installing the tools in the Prerequisites, navigate back to the JavaScript folder directory and type yarn. To build the WASM components, then type:

```
mkdir build
cd build
emcmake cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE
_C_FLAGS_RELEASE="-DNDEBUG -O3" -DCMAKE_SHARED_
LINKER_FLAGS_RELEASE="" ..
make
```

To build the webpack components, type yarn run build.

A.3.2 Basic Test

To run an example (1 server and 1 client) the following steps must be followed.

1. Open a terminal at the JavaScript folder, run the Server with the command $python -m http.server 9000 (the server starts listening).
2. Open another terminal at the Python server, run the application with the command $python3 src/app.py $<$symbol-size$>$ $<$wait-time$>$ e.g. $python3 src/app.py 100 40
   The symbol-size refers to the number of symbols/stocks, and wait-time refers to the time till the server waits (in seconds) to start the matching process. This means that the client has submitted 100 symbols to be matched (or not) with the symbols of the Server.
3. Next in a browser, go to http://localhost:9000/dist/ and answer the prompts with the symbol-size = 100 (should match the symbol-size we entered above) and the 1 (Short) or 0 (Long) direction/side. Short is for selling stock, and Long is for buying a stock. The hard-coded direction for the server is *Long*.
4. Initially, the client browser shows the table of symbols with randomly assigned quantities and with 0 as Matched; however, when the wait-time ends, the Matched will be updated after matching the symbols with the symbols of the server. This will conclude a successful execution.

A.4 Evaluation workflow

A.4.1 Major Claims

In this section, we provide the major claim of our paper. For testing purposes, we have created a UI that differs from the one in Figure 5 of [1], as the UI in effect is found in J.P. Morgan's markets portal, where Prime Match is deployed. The markets portal has been built over many years, and it is not available since it can reveal private information from other applications about the bank which are not relevant to this paper. The same holds for the trade management platform. In particular, our code in production was integrated into the trade management platform of the US bank, as described in detail in the paper (See Figure 6 for our final architecture). Table 2 of our paper is generated by running the repository we have uploaded on the private GitHub. We summarize our claim as follows:

(C1): *Prime Match has a throughput of 10 matches per second. This is proven by the experiment (E1) (See Section A.4.2) described in Section 5 of our paper, whose results are illustrated/reported in Table 2 of our paper.*

A.4.2 Experiments

Our experiment is described in Section 5 in the second paragraph of "Secure Minimum Protocol Performance".

(E1): *[25 human-minutes + 10 compute-minutes + <500MB disk]: This experiment aims to run an experiment with two clients and verify the running times as presented in Table 2 for 100, 200, 500, and 1000 symbols (for bank-to-client). This is done by adjusting the* symbol-size *in the command of Step 2 in Section A.3.2 per client. The code can also run for a larger number of symbols, such as 2000, 4000, etc. If one tests these cases, they should ensure that the* wait-time *is increased to a range of 50 to 100 seconds such that there is enough time to register the symbols of the clients before the matching process starts.*

Preparation: *Install the packages as described in Section A.3.1.*

Execution: *Follow all steps in Section A.3.2 but repeat Step 3 two times (sequentially) to accommodate a second client. Note that both clients must be initiated before the* wait-time *is passed. Moreover, the second client needs to be executed in incognito mode from the browser. Repeat the process for different numbers of symbols.*

Results: *After the completion of the experiment, right-click on the client browser to select* console *and check the running time and the MB sent and received, which are reported in the third, fifth and sixth columns of Table 2, respectively.*

References

[1] Antigoni Polychroniadou, Gilad Asharov, Benjamin Diamond, Tucker Balch, Hans Buehler, Richard Hua, Suwen Gu, Greg Gimler, and Manuela Veloso. Prime match: A privacy-preserving inventory matching system. Cryptology ePrint Archive, Paper 2023/400, 2023. https://eprint.iacr.org/2023/400.