Secure Distributed Matrix Multiplication

6.5610 Final Project, Spring 2024

Lila Chen, Fisher Jepsen, Serena Li, Younghun Roh

MIT

Introduction

Matrix multiplication is used extensively in many engineering fields and applications. As data and matrix sizes grow larger, naturally we would like to distribute the load of matrix multiplication across multiple servers. In the cases where the data is sensitive or we do not trust the computing servers to act honestly, clients may want cryptographic guarantees that a matrix multiplication protocol will hide the contents of one or both of the matrices involved.

In this paper, we will provide a survey of the types of matrix multiplication scenarios, the overall techniques used to address these scenarios and the changes that happen when we assume different things about the adversary's capabilities. Additionally, we will review methods to handle stragglers in these distributed systems.

0.1 Matrix multiplication scenarios

In general, the setup of the systems we discuss will be of one client and multiple servers. There are multiple settings for this system that one can consider.

1. Workers share library(s) of matrices. The identity of a matrix selected from a library by the client is kept private.
   (a) The client selects both matrices from the workers libraries.
   (b) The client selects one matrix from the workers' libraries, and the other is provided by the client. The matrix provided by the client is kept secret from the workers.
2. Workers do not share library(s) of matrices.
   (a) One matrix is made public to the workers, and the other is kept secret from the workers.
   (b) Both matrices are kept secret from the workers.

1 Private computation techniques

There are multiple cryptographic techniques that allow computation on data while maintaining privacy. In this section we will outline and compare three commonly used and well researched methods – additive secret sharing, polynomial secret sharing, as well as homomorphic encryption.

1.1 Additive secret sharing

Additive secret sharing is when the secret value is divided into shares such that a number of the shares can be summed together over a finite field to recover the secret. Since the shares are chosen randomly, an insufficient number of shares is not enough bits of information to recover the secret.

Additive secret sharing for n parties typically requires that all n shares are pooled together to recover the secret. Additionally, every share except one (the one set to the secret minus the sum of the other shares) is chosen randomly [15]. As we will see, polynomial secret sharing is not constructed in this way.

Another related technique is Beaver triples, which we used in our implementation. The Beaver triples method allows for multiplication of shares [15]. If two parties have shares x and y, and they want to compute the product of their shares $xy = z$. To do this, we can create a Beaver triple a, b, and c. a and b are uniformly random, and $c = ab$. Parties only have a share of the Beaver triple. Each party will publish their share of $x - a$ and $y - b$, and then individually compute z. Parties can compute z by the following

$$z = c + x(y - b) + y(x - a) - (x - a)(y - b)$$

$$z = ab + xy - xb + yx - ya - xy + xb + ay - ab$$

$$z = xy$$

Since a and b are uniformly random, parties only ever hold a random share of a and b, and parties only ever publish $x - a$ and $y - b$, none of the parties learn more information than they should. So this protocol is information-theoretically secure for the inputs [15].

1.2 Polynomial secret sharing

Polynomial secret sharing is when the secret value is set as the constant term of a polynomial. Then, the shares are set as the value of this polynomial at a specific point. The secret can be reconstructed with only a sufficient number of shares by polynomial interpolation [11]. The goal is to use these shares to find the low degree polynomial that passes through all of the points given by the shares. Once the polynomial is reconstructed the secret can be recovered.

Polynomial secret sharing allows for a threshold number of t shares to together recover the secret [11]. This might be more useful in situations where someone wants to let any two out of three trusted parties know the secret, for example, rather than all parties as in the case of additive secret sharing.

1.3 Homomorphic encryption

Homomorphic encryption is a type of encryption that allows for computations to be performed on data without decrypting it. It allows for storage and computation that is private and secure. Under the umbrella of homomorphic encryption, there is partially homomorphic encryption, leveled fully homomorphic encryption, and fully homomorphic encryption.

Partially homomorphic encryption involves computation on encrypted data with circuits that can only support one kind of gate. For example, the encryption scheme might be additively homomorphic or multiplicatively homomorphic, but not both. Leveled fully homomorphic encryption schemes, on the other hand, support arbitrary circuits but with bounded depth [8].

Lastly, and perhaps most importantly, fully homomorphic encryption allows computation under arbitrary circuits of any depth [9]. Clearly this is the strongest kind of homomorphic encryption. For several decades, the idea of FHE had been proposed but a robust scheme had not been described. Then, Craig Gentry at Stanford presented a bootstrapping method to turn a leveled homomorphic encryption scheme into a fully homomorphic one [7].

Fully homomorphic encryption schemes are not quite performant enough for practical use. Given even the theoretical scheme is so recent, implementations of it are still incredibly inefficient, producing large keys and ciphertexts. While fully homomorphic encryption has huge potential, its applications are still waiting to be unlocked.

2 Threat models

Distributing large matrix multiplication across a collection of servers we own is an ideal setting, because we already trust and control the computers performing the computations.

When outsourcing computation, we do not control the machines our calculations are performed on and must deal with external parties. Frequently there are adequate financial incentives for the external parties to at least behave honestly, i.e. provide correct results and follow an agreed upon protocol. However, the sensitivity of the data being computed upon may cause a need for stricter guarantees on the information disclosed to collaborating parties. Businesses may trust the financial incentive for correct results but do not want to reveal their intellectual property to other parties. Healthcare providers have strong requirements for not revealing patient data and for correctness in computation results.

Below are two different kinds of threat models, which describe some assumptions that researchers make about the kinds of threats to correctness and privacy. In all these threat models, the goal is to ensure that workers learn nothing about the data being computed on.

1. **Semi-honest:** Some workers may collude and share information they learn throughout the protocol. However, workers do computations correctly and return correct results.
2. **Malicious:** Some workers may collude and share information they learn throughout the protocol. Workers may return incorrect outputs or not follow the protocol.

We typically handle semi-honest workers through some transformation on the client's inputs. The cost of pre-computation by the client and the total communication cost are the main metrics used to judge utility of semi-honest schemes. Most semi-honest schemes are based on somewhat homomorphic encryption, do not include output validation and compute inner products [14] [6]. Other semi-honest schemes for outsourcing computation can be done with relatively simple permutations on the input matrices.

The malicious setting presents additional challenges, as we must verify worker or protocol outputs. The computation cost of verifying each parties outputs should not be greater than the cost of performing the computation locally. The malicious setting has two sub-classes. The first is a *honest majority*, where the majority of participants are trusted to behave honestly, i.e. if n parties are required to compute some output, less than $n/2$ parties behave maliciously. The second is *dishonest majority*, where a majority of parties behave maliciously. The strongest setting of this is $n-1$ (all-but-one) parties behaving maliciously.

2.1 Semi-honest works

Jiang [10] introduces an efficient encoding of matrices and evaluation methods for computing matrix operations under homomorphic encryption. Its key contribution in this space is in its representation of the matrix into *packed ciphertexts*, where instead of naively encrypting each row vector of a matrix, many rows are packed together, and can thus reduce the total number of homomorphic operations needed to multiply the matrices. However, it only satisfies the semi-honest setting, as no verification is performed on the end result.

2.2 Malicious works

Lei [13] proposes a relatively elegant scheme to outsource multiplication of two matrices to other machines. It is permutation based, unlike most of the other inner-product based schemes. It is designed for a single client-server pair, but due to being permutation-based, can be distributed amongst many machines using normal non-secret distributed matrix multiplication methods. The inputs X (of size $m \times n$) and Y (of size $n \times s$) are hidden by generating three secret, invertible, and randomly-permuting matrices $P_{1..3}$. The client computes $X' = P_1 X P_2^{-1}, Y' = P_2 Y P_3^{-1}$, and provides (X', Y') to the server. The server computes $X' \times Y'$ as normal, and provides the result $Z = X'Y'$. The client finds the original answer through $X \times Y = P_1^{-1} Z P_3$. The precomputation of X' and Y' can be done in $O(n^2)$ time, and verification can also be done in $O(n^2)$ time. Dropping verification in the semi-honest setting makes this one of the most efficient methods for the outsourcing-computation problem.

Chen [3] is an inner-product-based work using somewhat homomorphic encryption, similar to Jiang [10], but adds the SPDZ framework [5] to allow for MPC, secure in the dishonest majority setting. Beaver triples are used to allow for multiplication of shares between many parties, and alone are sufficient for MPC matrix multiplication in the semi-honest setting. Chen utilizes the SPDZ framework to support *authenticated* beaver triples. In addition to being given shares of values to multiply x, parties are given shares of a global MAC key α and a MAC share m, where $\alpha \cdot x = m$. These MACs are additive when multiplying shares, allowing for the end of each inner product to be authenticated. This scheme has both a high computation and communication cost compared to non-homomorphic/non-MPC works like Lei2014, but is state-of-the-art in its category.

3 Dealing with stragglers

There is also a body of work proposing methods to deal with stragglers in these distributed matrix multiplication systems. Often, these protocols involve encoding the matrices in such a way that answers may be reconstructed without needing all responses from the computing servers. Many of these protocols define the following parameters of their algorithms:

1. Recovery threshold: The minimum number of worker responses needed before the client can reconstruct the answer.

2. Collusion: The maximum number of workers that can collude without leaking information about the secret matrices.

3.1 Zhu2023

Zhu2023 [17] proposes protocols that reduce the effects of stragglers for two types of matrix multiplication problems. The first problem they consider is that of "Private and Secure Matrix Multiplication", where a library of matrices is stored on the computing server, and the client has a private matrix A that it would like to multiply against one of the matrices from the library. The client would like to keep both A and the identity of the library matrix private from the servers.

The second problem is that of "Fully Private Matrix Multiplication", where both matrices are selected from two libraries of matrices shared among the computing servers. The client would like to keep the identity of these two matrices secret from the servers.

Both problems are considered in the setting of honest-but-curious servers, and the protocols they provide have information-theoretical privacy for the identity and contents of the matrices. To achieve resilience against stragglers, they utilize MDS-coded storage via Reed Solomon codes such that each matrix in a library can be reconstructed from at least K out of the N servers (thus tolerating $N - K$ server failures). This unique property of the storage allows them to construct a protocol in which only P out of N servers must return a response before the answer can be reconstructed.

3.2 Kim2019

Kim2019 [12] introduces a protocol based on polynomial codes for the problem of distributed private matrix multiplication in the setting where one matrix A is provided by the client and the other matrix B is selected from a library of matrices shared by the workers. The protocol achieves security when no workers collude, and has a recovery threshold based on the number of submatrices A and B are divided into.

A is divided into m submatrices and B is divided into $n - 1$ submatrices, such that the final product AB can be recovered from the products of the submatrices. Then, they construct a polynomial from the submatrices of A and to each worker, send an evaluation of this polynomial at a random point.

Each matrix in the workers' library of matrices is also used to construct a polynomial. For each matrix that is not B, evaluations of its polynomial at random points are sent to the workers. For B, the same evaluation of its polynomial is sent to all the workers. Since this protocol assumes that no workers collude, the workers cannot realize that B is the target matrix.

Each worker now computes a function on their assigned evaluations of the matrices' polynomials, and returns their result to the client. The client can then reconstruct the final product once $mn + n$ responses are received.

4 Applications

There are increasingly many possible applications for private matrix multiplication, which we will discuss in this next section. We will touch on possible implementations of private matrix multiplication to machine learning, secure databases, auctions and voting, as well as genomic analysis.

As we improve the field of machine learning, the data sets and models used grow much larger. New models are currently being developed that produce more high-level and complex outputs, at the cost of requiring more detailed and specific inputs. While it is natural to apply these machine learning models to our own society in order to learn and make better predictions about ourselves, it means that very large amounts of data describing our society is necessary to train the models. This raises the serious question of privacy–who is this data coming from, and did they consent to its use in this setting?

Private matrix multiplication is a solution to this problem [3]. Matrix multiplication underlies machine learning. Instead of publicizing data sets to train models, to prioritize privacy we can use these methods to obscure data sets from the models. Machine learning with private matrix multiplication has numerous applications–while current state-of-the-art models are trained on the entirety of the open Internet, expanding the available data to train models would greatly improve the quality of the models. The technique of private matrix multiplication makes a host of other data usable, such as from the census, as well as finance, legal, education, or healthcare sectors. Any proprietary data also becomes usable.

This approach has particularly advantageous applications to healthcare. Say two hospitals have information about their patients. Due to privacy laws, they are not permitted to disclose any of this data. However, they would like to combine forces to produce a model that would allow them to improve their patient experience. With private matrix multiplication, each hospital can keep its patient data from the other, but they can still achieve the results they want. This could be through applying these techniques to federated transfer learning frameworks, for example [4].

We can also apply it to genomic analysis and biometric identification [16]. As noted in this paper, private matrix multiplication enables systems to use personal data, such as fingerprints or face scans, to uniquely identify users. This can be very useful in the authentication setting, as it is reasonably difficult for an adversary to forge this biometric information. However, the challenge is that once the adversary has collected the data then they can use it to impersonate a victim on many different platforms. As such, when using biometric identification it is particularly crucial that the data is private, which this setup allows.

Lastly, for similar reasons, it can also be used towards implementing auctions and voting [2]. In an auction setting, bids need to remain confidential. Private matrix multiplication can allow sensitive information to stay hidden, preventing collusion and maintaining fairness. With voting, the same is true. Preserving anonymity and integrity is critical to holding a representative vote and a transparent election. The potential applications of private computation extend far beyond matrix multiplication, encompassing a wide range of scenarios where preserving privacy is essential.

5 Our implementation of private matrix multiplication

To show that Secure Distributed Matrix Multiplication is actually feasible in practice and to understand the difference in security settings in depth, we implemented a simple version of Secure Distributed Matrix Multiplication.

Our implementation is public in https://github.com/Diuven/dispense, written in C++ and Javascript. Core logics are in EntryServerHandler class of src/entryServer.hpp.

5.1 Problem Settings and System Overview

We assume the following settings for the problem:

1. The client has input matrices $A, B \in \mathbb{Z}^{N \times N}$, and keeps them secret from the workers. (2-a in section 0.1)
2. Each matrices and their multiplication results are in the range of 64bit integers.
3. Workers are semi-honest. They will give the correct results, but may try to infer the input matrices. Workers may collude with each other.

In overview, the system works as follows:

1. The client opens a websocket server, and waits for the workers to connect. Upon connection, each worker is assigned to a unique ID.
2. Client generates two random matrices $A, B \in \mathbb{Z}^{N \times N}$, and generates a list of tasks with random task ID. Each task will be a pair of vectors.
3. When the client starts, workers request to be assigned a task, request the first vector of the task, and request the second vector of the task.
4. After getting the full task from the client, workers calculate the dot product of the vectors, and send the result back to the client.
5. Client looks up the task ID to check the original location, and aggregates the results to the corresponding location of the result matrix.
6. When all tasks are consumed, the client closes all connection and returns the result matrix.

We implement multiple security measures during the task generation step, to ensure the privacy of the input matrices.

- Utilizing Beaver triples to obscure the input matrices.
  - The client (uniformly) generates random S pairs of vector $(\text{amask}_i, \text{bmask}_i)$ from $\mathbb{Z}^N$, and precomputes the dot product of the pairs, $\text{offset}_{i,j} = \text{amask}_i \cdot \text{bmask}_j$.
  - Generate two tasks for each (i, j) pair of the input matrices: $(X_{i,j,+}, Y_{i,j,+}) = (A_{i,*} + \text{amask}_{r(i)}, B_{*,j} + \text{bmask}_{r(j)})$ and $(X_{i,j,-}, Y_{i,j,-}) = (A_{i,*} - \text{amask}_{r(i)}, B_{*,j} - \text{bmask}_{r(j)})$, where $r(i)$ is a random function of $[N] \to [S]$.
  - Each entry of the result matrix R will be $2 \cdot R_{i,j} = (X_{i,j,+} \cdot Y_{i,j,+}) + (X_{i,j,-} \cdot Y_{i,j,-}) - 2 \cdot \text{offset}_{r(i),r(j)}$.
  - This prevent workers from inferring the exact elements of the input matrices.
- Shuffle the order of tasks and randomly assign the tasks to workers.
  - Prevent workers from inferring which position the vectors are in the input matrices.
- Randomly permute the input matrices.
  - Prevent workers guessing the Beaver triple pairs and sum up to guess the original vector.
  - Note: we removed from this step from our implementation due to performance issues.
- Randomly swaps the order of two vectors in each tasks.
  - Prevent workers from inferring whether a vector is from A or B.

For convenience of understanding, let's suppose that N is 1024 and S is 16. Note that S should be significantly smaller than N, since we are precomputing the dot product of the pairs.

5.2 Analysis and Potential Improvements

5.2.1 Security Analysis

To evaluate how secure our system is, we will analyze the probability of workers correctly inferring the input matrices, in different scenarios. For convenience, let's denote the finite base field as $\mathbb{F}$, and number of clients are fixed to M.

Assume a worker got only one task, and it does not collude with any other workers. Since each elements are obscured by uniform vector addition (amask and bmask), the worker cannot tell difference from the uniform distribution. Therefore, their guess is as good as random guess, and the probability of guessing correctly is $1/|\mathbb{F}|^{2N^2}$. In our setting, $|\mathbb{F}|$ is 2^{64}, and N is 1024, so the probability is about $2^{-2^{27}} \approx 2^{-1.3 \cdot 10^8}$.

Assume a worker got two tasks, and it does not collude with any other workers. If the tasks are from the different rows and columns, then the worker will have no information about the input matrices, by the same reason above. However if at least one of the vectors are from the same row or column with different Beaver triple sign, then the worker can try to unshuffle and sum up the vectors to guess the original vector.

- Probability of a worker to get two tasks from the same row or column with different sign is $\frac{2N-1}{2N^2-1} \approx 1/N$.
- Probability of choosing the correct pairs to add given two pairs is about $1/4$. (ignoring edge cases with negligible probability)
- Probability of unshuffling the two vectors correctly is $1/(N!)^2$.
- Probability of guessing which matrix and position the vector is from is $1/2N$.
- Probability of guessing the other vectors right is $1/|\mathbb{F}|^{2N^2-N}$.

Therefore, the best guess of the worker will be correct with probability of $(1/|\mathbb{F}|^{2N^2}) \cdot (|\mathbb{F}|^N/(8N^2(N!)^2))$. In our setting, the improvement factor from the first scenario is less than 2^{1300}, which will still give $2^{-1.3 \cdot 10^8}$. So the probability is still negligible.

Assume all workers colluded and got all N^2 tasks. Then, first of all, they can learn what is the sum of all elements in the input matrices by adding all the tasks. Although this can be meaningful data of itself, it only provides 64 bits of information. To reconstruct full the input matrices, they need to guess the correct positions of the vectors, and the correct order of each elements in each vector.

For given row/column index and the Beaver triples sign, consider the set of elements in the corresponding vector as $R_{\alpha,i,\beta}$. For example, $R_{A,i,+}$ is the set of elements in the i-th row of A with the positive Beaver triple sign. It is very likely that each $R_{\alpha,i,\beta}$ is unique, which means the workers can group the vectors in different tasks.

Then, having $4N$ sets, they have to correctly assign and order the elements to reconstruct the input matrices.

- Probability of guessing the correct sign of each set is $2/{}_{4N}C_{2N}$
- For each sign, probability of guessing the correct matrix of each set is $1/{}_{2N}C_N$
- For each sign and matrix, probability of guessing the correct positions of the vectors is $1/N!$.

- For each vector, probability of guessing the correct order of the elements is $1/N!$.

The resulting probability is $(2/{}_{4N}C_{2N}) \cdot (1/{}_{2N}C_N)^2 \cdot (1/N!)^{4N}$, which is about $2^{-3 \cdot 10^7}$. If we skip the permutation for each vector, the probability is $(2/{}_{4N}C_{2N}) \cdot (1/{}_{2N}C_N)^2$, which is about 2^{-7200}.

Note that we did not take into account that we limit the number of Beaver triples, S, to 16. This might provide some extra bits of information, but it will significantly complicate the analysis.

5.2.2 Performance Analysis

Currently, communication cost of the client is $O(N^3)$, since we send each vectors $2N$ times, where each vector have N integers.

This is undesirable, since the conventional matrix multiplication can be done in $O(N^{2.3})$. This implies we are spending more resource for communication than what we would have spent for the computation itself. We expect to reduce this by batching in the future works.

Time complexity for the client is $O(N^2 + NS^2)$, since the client will only distribute and aggregate the tasks, and do the dot product of S^2 Beaver triple masks. The overall time complexity as a system is $O(N^2 + N^3/M + NS^2)$, since the computation is distributed to M workers.

Our implementation has successfully handled 27 workers during the presentation, and it took more than 5 minutes to complete $N = 1024, M = 4, S = 16$ setting. For comparison, the single device with single thread takes less than 10 seconds, and single device with 8 threads takes less than 3 seconds.

5.2.3 Potential Improvements

Currently, our scheme does not handle colluding or malicious workers. It would be an improvement to handle these, as our scheme would be more robust against dishonest parties. One way to handle colluding parties could be to use a t-out-of-n polynomial based secret sharing scheme, which would be resistant to up to t parties colluding. Using this instead of additive secret sharing and Beaver triples would allow a threshold t parties to recover the secret. Another possible technique we could use, as outlined in Section 1.3, would be homomorphic encryption. We could combine Beaver triples with linearly or even fully homomorphic encryption to use in our implementation. Despite perfomance tradeoffs, this might be worthwhile for further exploration.

Currently, our algorithm leaks information if a worker is assigned two different shares of the same row or column. We can improve this by requiring that workers will receive at most one share of each row or column.

Our scheme is not yet resilient against stragglers, as the client will only stop sending tasks to a worker once the client-worker connection is closed. One way to improve this would be banning stragglers which take too long to respond. Alternatively, we could utilize polynomial codes as described in the Stragglers section above.

Lastly, we could implement a class of improvements that make the performance of our implementation better. This would generally involve optimization and engineering techniques, such as bathcing, pipelining, using multicore workers, better serialization, or even splitting the matrix multiplication more efficiently. For example, using block partitioning of the input matrices would allow us to handle cases of extremely large matrices with less memory needed on the clients.

6 Conclusion

In this paper, we have provided a survey of distributed and secure matrix multiplication methods and a proof of concept implementation of one such method. We reviewed four different settings of this matrix multiplication problem, including settings where the workers share a library of matrices to ones where they do not. Next, we discussed different techniques used by researchers to construct these matrix multiplication schemes, including additive secret sharing, polynomial-based secret sharing, and homomorphic encryption. We also provide a comparison of different threat models considered by protocols, and review a few of the papers in the semi-honest and the malicious settings. The final part of our survey examines how stragglers in distributed systems can affect the design of matrix multiplication schemes, and covers some protocols based on polynomial codes that solve this problem. We also provide an implementation of a solution to one problem setting of distributed and private matrix multiplication, demonstrating its feasibility in practice.

7 Acknowledgements and Member Contributions

We would like to thank Prof. Corrigan-Gibbs, Prof. Kalai and the 6.5610 teaching assistants for their support. Individual member contributions are listed below.

- Lila Chen: Private matrix multiplication techniques and applications.
- Fisher Jepsen: Threat models considered in private matrix multiplication.
- Serena Li: Types of private matrix multiplication problems and stragglers in distributed systems.
- Younghun Roh: Implementation and analysis of private matrix multiplication.

References

[1] Malihe Aliasgari, Osvaldo Simeone, and Jörg Kliewer. Private and secure distributed matrix multiplication with flexible communication load. *IEEE Transactions on Information Forensics and Security*, 15:2722–2734, 2020.

[2] David W. Archer, Dan Bogdanov, Liina Kamm, Y. Lindell, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. From keys to databases – real-world applications of secure multi-party computation. Cryptology ePrint Archive, Paper 2018/450, 2018. https://eprint.iacr.org/2018/450.

[3] Hao Chen, Miran Kim, Ilya Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. Maliciously secure matrix multiplication with applications to private deep learning. Cryptology ePrint Archive, Paper 2020/451, 2020. https://eprint.iacr.org/2020/451.

[4] Yiqiang Chen, Jindong Wang, Chaohui Yu, Wen Gao, and Xin Qin. Fedhealth: A federated transfer learning framework for wearable healthcare. *CoRR*, abs/1907.09173, 2019.

[5] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[6] Dung Hoang Duong, Pradeep Kumar Mishra, and Masaya Yasuda. Efficient secure matrix multiplication over lwe-based homomorphic encryption. *Tatra mountains mathematical publications*, 67(1):69–83, 2016.

[7] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[8] Alexandra Henzinger. Mit 6.5610 lecture 8 notes - fhe (part 1), 2024. https://65610.csail.mit.edu/2024/lec/l08-fhe.pdf.

[9] Alexandra Henzinger. Mit 6.5610 lecture 9 notes - fhe (part 2), 2024. https://65610.csail.mit.edu/2024/lec/l09-fhe2.pdf.

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

[11] Yael Kalai. Mit 6.5610 lecture 15 notes - secret sharing schemes, 2024. https://65610.csail.mit.edu/2024/lec/l15-ss.pdf.

[12] Minchul Kim and Jungwoo Lee. Private secure coded computation. *CoRR*, abs/1902.00167, 2019.

[13] Xinyu Lei, Xiaofeng Liao, Tingwen Huang, and Feno Heriniaina. Achieving security, robust cheating resistance, and high-efficiency for outsourcing large matrix multiplication computation to a malicious cloud. *Information sciences*, 280:205–217, 2014.

[14] Lihua Wang, Yoshinori Aono, and Le Trieu Phong. A new secure matrix multiplication from ring-lwe. In *Cryptology and Network Security: 16th International Conference, CANS 2017, Hong Kong, China, November 30—December 2, 2017, Revised Selected Papers 16*, pages 93–111. Springer, 2018.

[15] David Wu. Stanford cs355 lecture 7 (4/23/18), 2018. https://crypto.stanford.edu/cs355/18sp/lec7.pdf.

[16] Dongyu Wu, Bei Liang, Zijie Lu, and Jintai Ding. Efficient secure multiparty computation for multidimensional arithmetics and its application in privacy-preserving biometric identification. 2023. https://eprint.iacr.org/2023/1863.

[17] Jinbao Zhu, Songze Li, and Jie Li. Information-theoretically private matrix multiplication from mds-coded storage. *IEEE Transactions on Information Forensics and Security*, 18:1680–1695, 2023.