SecRec - A Secure Recommendation Algorithm

Siddharth Diwan
Brown University

Edward Bielawa
Brown University

William Sun
Brown University

Abstract

Many services use recommendation algorithms to provide suggestions on products or items that users are likely to enjoy given their previous actions. However, the method in which typical recommendation algorithms decide what recommendations to provide to users requires leaking information about users' previously shown preferences, and this information can be used to deanonymize users in the system, either maliciously by the server itself or via a breach of the server by a malicious actor.

This paper describes SecRec, a recommendation system that can provide the privacy guarantee that a server running SecRec cannot learn which users provided what information or what data users are requesting from the server, information that is normally required to perform such recommendations. We show SecRec's efficacy using both a standard matrix completion recommendation algorithm and a more complex robust matrix completion algorithm, and show that the system preserves the efficacy of the algorithms, despite incurring a significant runtime overhead due to the time cost of using a fully homomorphic encryption (FHE) scheme. The code for SecRec can be found at https://github.com/sidwan02/SecRec.

1 Introduction

Recommendation algorithms are commonplace in frequently used services, from search engines such as Google to movie viewing services such as Netflix. They are crucial in allowing services to provide users with tailored recommendations to users by predicting what users would like given what they have previously shown interest in. Based on information that the users of the service provide or that is collected when a user is using the service, these recommendation algorithms can predict whether a user is likely to enjoy a particular recommendation or not.

However, the information collected by services using recommendation algorithms can be sensitive personal information, and as a result, can potentially be used to de-anonymize users. For example, in order for Google to give search results to users, they collect not just the query itself, which can contain personal information, but also ask for information such as location and prior search history [5]. Netflix, which only needs to recommend a sorted list of potential movies to watch, also collects information such as engagement or other interactions with the Netflix application [4]. In addition, outside of interaction-based data, simply knowing what ratings a given user has given some set of movies can be enough to de-anonymize a user. This becomes a problem when considering that the services collecting this information can leak this information, either inadvertently through data breaches or intentionally by selling the data to advertisers.

In this report we present SecRec, a recommendation system that provides the privacy guarantee that the server cannot know what information the user provided to the server, whether that is information provided to allow the recommendation algorithm to train or information the user provides to request recommendations. We use the simple paradigm of the movie recommendation problem, where users provide movie ratings for the movies they have seen, and the system must predict ratings for movies users haven't seen.

We develop a scheme in which the server performs computations over data that is homomorphically encrypted, such that the server cannot learn the true information in the data that it is computing over. The rating matrix is encrypted with this scheme, which allows for matrix completion to be done without the server knowing what is in the matrix. In order to allow the matrix completion algorithm to know when it is complete, we additionally create a matrix representing whether an element in the rating matrix is a true value or a predicted value, and we expose to the server a way to decrypt just the truth value of a comparison to verify whether a value is true or predicted. We also implement private information retrieval (PIR), which allows for users to request ratings from the server in a way that does not expose to the server what the user is requesting.

We then evaluate this system on a test sample of randomly generated user rating data. We find that the secure versions of the recommendation algorithms achieve similar loss curves

as the equivalent non-secure versions, and have similar matrix norms, which indicates that both algorithms have roughly equivalent accuracy in completing the matrix entries. We also find that a main limiting factor of this system is runtime, as the secure recommender runs upwards of 100,000 times slower than the equivalent insecure recommender, and scales poorly with larger matrices. However, we suggest that this system serves a proof of concept that recommender systems can be made secure, and additionally offer potential routes of optimization for the future.

2 Background and Related Work

This work is closely related to the work of Tiptoe [3], which describes a system for private internet search in a scenario when the server is untrusted. We build off of Tiptoe's idea of using fully homomorphic encryption and private information retrieval to allow the server to do intense computations while ensuring that the server cannot know information about the user. We adapt this to the related but different scenario of recommendation algorithms. Notably, recommendation algorithms use information about users to actually compute recommendations, rather than simply looking for semantic similarity like the web search algorithm discussed in Tiptoe, and so further security is required to ensure recommendation systems do not leak information.

This work is also similar to the work of PrivateKube [6], which makes users' information private in a scenario where user information is used to train models. PrivateKube handles the threat model that when user information is used to train a model, that model can also leak information about the users whose information is used to train the model. PrivateKube is concerned with the potential that if a user's data is used too often to train data, a malicious actor can expose the information from models with that user's data, and thus it uses differential privacy (DP) methods to prevent information from being leaked from those models. We build off of the similar goal of enforcing the prevention of information leaking from models via privacy guarantees. However, the work of SecRec differs from that of PrivateKube in that it uses a fully homomorphic scheme rather than differential privacy, and the threat model of SecRec is less concerned with a malicious user trying to extract information from the output of a model and is instead concerned about trusting the server running training computations.

3 Design

3.1 SecRec Architecture

An overview of the architecture of SecRec is shown in Figure 1. It consists of 3 components: the user, the combiner, and the server, which each implement a different part of the recommendation system.

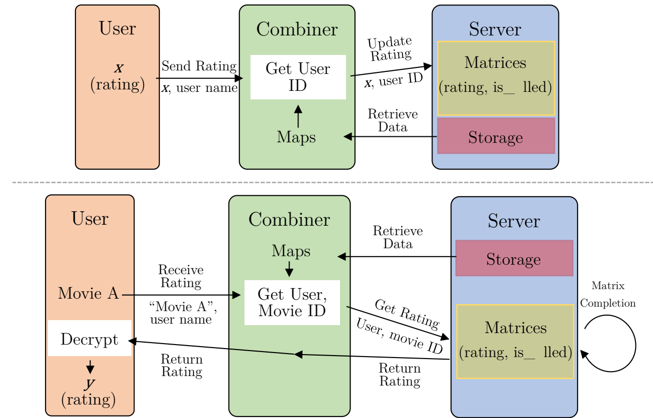


Figure 1: The SecRec component architecture. The top half depicts the process by which an update to a rating for a movie would occur, and the bottom half depicts the process by which a user would request a new recommendation from the recommendation algorithm. The yellow region represents information encrypted by FHE, while the red region represents information encrypted via hybrid encryption.

3.1.1 Core Challenges and Threat Model

We are concerned with the possibility of a malicious server, and so the server component is assumed to be untrusted. As such, everything in the server that pertains to potentially private user information needs to be encrypted to ensure that the server cannot learn information about those users. However, recommendation algorithms are computationally expensive, even in the most simple of cases, and often require a large amount of user data to get high-quality recommendations, which means that as much of the computation should be performed on the server. This is the challenge that we attempt to tackle using SecRec.

Note that we assume that things run on the user's end are trusted, meaning we trust the user's device, including the CPU and storage the user uses. In addition, we are not concerned with side-channel attacks that the server can use to leak information without directly accessing the data, and instead focus on protecting from the server directly accessing the private user data.

3.1.2 The User Component

The user component represents the main trusted portion of the SecRec system, and this component has 2 main goals: to be able to send ratings to the server to update the recommendation algorithm, and to be able to ask for recommendations. To do this, the user stores a private/public key pair which it can use to encrypt information to send to the server, and then leverages the APIs provided by the combiner (described in further detail in Section 3.1.3).

When the user wishes to send a rating to the server, it will encrypt the rating, and call the API provided by the combiner, along with the user's own name. In addition, in this scenario we represent the user asking for a recommendation via the user asking for the algorithm's predicted rating of a particular movie for this user. In such a scenario, when the user wishes to request a recommendation it will send the name of the appropriate movie and user to the combiner API, and receives an encrypted predicted rating, which it can decrypt using the private key it has.

3.1.3 The Combiner Component

The combiner is a trusted component that serves as an API for the user to call to perform sending/retrieving rating functionalities. The combiner is responsible for processing the data given by the user, sending it to the server, and if necessary, receives the resultant value from the server, performs some post-processing, and then sends it back to the user, such that the user does not have to handle specifics about the server.

As a result of the combiner being the trusted component to directly interface with the server, it is the component that performs all the computations and look-ups relevant to private information that the user is not given to the user (to simplify the job of the user). In our scenario, this manifests in our combiner implementation requiring 2 maps:

- `user_ownership` is a map from $\{u_i : \texttt{row_id}\}$ where the key u_i with $i \in \{0, ..., n_u - 1\}$ is the unique username and the value is the row index of the recommendation matrix stored by the server.

- `movie_ownership` is a map from $\{m_j : \texttt{col_id}\}$ where the key m_j with $j \in \{0, ...n_m - 1\}$ is the unique movie name and the value is the col index of the recommendation matrix stored by the server.

These are both pieces of information that should not be known by the server since it can leak private information, so the combiner would need to be the only one with access to this. However, this can grow very large depending on the number of users, and we aim to limit the amount of storage this system requires on the user's end. As such, the combiner also leverages server storage, which is a system by which the combiner stores data on the server, and then retrieves it when necessary from the server to use it. Since the server is untrusted, this storage is encrypted; this is further described in Section 3.2.

3.1.4 The Server Component

The server is our untrusted computation unit in SecRec, and we want to perform as many intensive operations on the server as possible without leaking information. As such, the server is the component that runs the matrix completion algorithm - more specifics of the algorithm we use are described in Section 4.2.

To perform matrix completion, the server stores the matrix relevant to our recommendation algorithm - in our scenario, this is a matrix where the rows are users, the columns are movies, and the entries are ratings of that user for that movie. These entries may either be real ratings that the user provided, or predicted ratings computed by the matrix completion algorithm, and knowing which is which is crucial for the matrix completion algorithm to run properly. To allow for this, the server also stores an is_filled matrix which is the same size as the ratings matrix and is populated with 1s and 0s to represent whether or not the element in that position in the ratings matrix is a real entry (1) or predicted (0). These matrices are encrypted so the server can't see the actual data in the matrices; the encryption scheme is described in Section 3.2.

In addition, as mentioned in Section 3.1.3, the server stores data for the combiner in its server storage. This is exposed via an API where the combiner can send bytes to the server with a certain key, and retrieve it via accessing the data at that key (i.e. a key-value store). This data is expected to be encrypted by the combiner itself, and so the server simply stores the data for the combiner.

3.2 SecRec's Encryption Scheme

As described in Section 3.1.4, the server is untrusted, so we need to make sure everything stored on or sent to the server is encrypted. SecRec's server contains 2 types of items that they may access or store: data the combiner sends to the server for server storage, and data relevant for matrix completion computations (such as the user or movie ID, or the ratings or is_filled matrix).

3.2.1 Matrix Completion Data Encryption

For items that pertain to matrix completion, these items need to be able to have computations performed on them in the server, since these computations are intensive and so we want to perform them on the server. This means they need to be encrypted using a fully homomorphic encryption (FHE) scheme, which is an encryption scheme where performing a mathematical operation on a ciphertext and then decrypting is equivalent to decrypting first and then performing the operation on the decrypted data.

We note here that for this implementation of SecRec, we have it such that all users must share the same key for the FHE scheme; however, this presents the challenge of ensuring that all users synchronize to have the same key, and in an environment where the server is untrusted, this proves to be difficult in practice, since we cannot trust the server to send the key. THFE (Threshold FHE) would solve this issue, but we did not have the time to explore an implementation of THFE, so it serves as future work for this system. We discuss this further in Threshold FHE.

Private Information Retrieval. Direct application of a FHE scheme works in most scenarios. However, when requesting a rating from the server, we cannot simply encrypt the user and movie ID and send them to the server, because we can no longer use that information as an index into the matrix to retrieve the rating if that information is encrypted, and we don't want to send them unencrypted, since knowing what the user is attempting to retrieve leaks information about the user.

To get around this, we implement a private information retrieval scheme (PIR), where we can retrieve information from a server without the server knowing what we are retrieving. Inspired by Tiptoe, we create 2 one-hot vectors that are encrypted with FHE. One of them has size equal to the number of users and contains all 0s except for the index of the user which has a 1. The other has size equal to the number of movies and contains all 0s except for the index of the movie which has a 1. Then, we perform matrix multiplication to multiply the rating matrix with the movie one-hot vector to get a vector of all the user ratings for the desired movie, and we then perform a dot product of that with the user one-hot vector to get the rating we want, all in an encrypted manner.

3.2.2 Server Storage Encryption

For data that the combiner stores on the server, this data does not need to have computations performed on it by the server; instead, the combiner needs only to be able to store that data on the server securely and retrieve it when it needs to. Thus, we do not need FHE for this encryption.

Instead, we encrypt our plaintext data using a symmetric encryption scheme. We choose symmetric instead of asymmetric encryption to support encryption of arbitrary length using chain-block-ciphers since asymmetric encryption schemes have an upper limit on the plaintext size they can encrypt. To secure the symmetric encryption key, since it can be used to both encrypt and decrypt the data, the combiner then encrypts the symmetric encryption key using asymmetric encryption and stores that on the server as well, finally storing the asymmetric encryption keys in a key database in the combiner. (This design choice is detailed further in).

However, since the server can theoretically access the public key for this encryption, if the server was malicious it could sign some other data and replace the data stored on the server, and the decryption on the combiner's side would succeed even if the data it now has is not the data the combiner had originally put into the storage. To this end, we also sign all the data going into server storage before it's encrypted, and store the sign and verify keys for this in the combiner's key database as well.

4 Implementation

4.1 Secure Wrappers

In our implementation of the SecRec system, when the server is first initialized, it is provided certain functions that we call "secure wrappers". While the server is not allowed explicit access to the private key of our encryption since the server is untrusted, these wrappers are used to securely perform decryptions using s_k in intermediate computation scenarios when decryption is necessary due to the limitations of FHE, and these wrappers re-encrypt the result prior to returning it back to the server.

Such decryption wrappers are required because in FHE, multiplications between ciphertexts are very expensive. Each multiplication increases the error of the encrypted value (amount of noise differentiating it from the true unencrypted value) exponentially. From our testing, one can only perform about five multiplications on ciphertext before this error grows larger than the true product, making the resulting computation useless. These secure wrappers are:

- `secure_matrix_error_reset_wrapper` decrypts and re-encrypts the matrix used in the matrix completion algorithm. Matrix completion (in particular matrix multiplication) on FHE data involves many multiplications on ciphertexts, which accumulates error. As a result, we must reset the errors of the ciphertexts stored in the matrix completion algorithm by re-encrypting the matrices.

- `secure_svd_wrapper` performs singular value decomposition (SVD) for the server. We experimented with different methods of implementing SVD from scratch to support FHE operations with SEAL, such as eigenvalue iteration and QR decomposition. Choosing an SVD algorithm involves trade-offs between algorithmic efficiency and ease of implementation. A feasible SVD implementation must only use addition and multiplication operations on encrypted data, which most standard SVD algorithms do not.

  We chose to implement an eigenvalue iteration algorithm, which iteratively computes eigenvalues (singular values) and eigenvectors of the input matrix. Although eigenvalue iteration is slower than most SVD algorithms, it is far easier to implement in a FHE scheme than alternatives. Other SVD algorithms require complex subroutines that make use of operations beyond addition and multiplication, or do not generalize correctly to rectangular (non-square) matrices.

  Our SVD algorithm requires normalization of eigenvectors during iteration. Although there exist methods to normalize vectors in an encrypted setting [1], these methods require many multiplications on ciphertexts, which causes large error, making such a normalization task not feasible for the FHE setting. As a result, our

implementation decrypts resulting eigenvectors for normalization while computing the SVD. Potential future work in improving the efficiency and security of SVD over encrypted data is described in Performing SVD under FHE.

- `secure_clip_wrapper` clips the predicted values given by matrix completion. In matrix completion, we would like to clip recommendations predicted by the algorithm to the range of $r \in \{0.5, 1.0, 1.5, ..., 5.0\}$. Performing inequality comparisons between FHE numbers can be approximated, but this approximation requires many encrypted multiplications [1]. This number of multiplications is not feasible due to the rate at which encrypted values accumulate error during multiplication. Any FHE clipping operation would require decrypting and re-encrypting the input values to sidestep error accumulation. Therefore, we choose to forgo a FHE implementation of clipping and instead perform clipping securely in a non-encrypted setting.

- `secure_division_wrapper` performs a division computation. FHE does not support division, but it is required in matrix completion to training losses and other metrics, which require division over the number of true user-input recommendations. Approximations for division can be implemented in a FHE scheme [1], but the resulting error from a division approximation combined with error accumulated from matrix completion makes this approach not feasible. Therefore, we opt to forgo a FHE implementation of division and instead provide a safe division wrapper that performs unencrypted division.

We note that this allows the server to learn a small amount of information about the data via learning about intermediate computation values. There are methods of limiting this, and we propose one in Privacy Budgets for Secure Wrappers.

4.2 Matrix Completion Algorithm

SecRec predicts ratings for users using a matrix completion algorithm, which is an algorithm that predicts unobserved entries in a semi-observed matrix based on the already observed data. There are two matrix completion algorithms used:

- Regular matrix completion, which uses a gradient-based matrix completion approach with alternate minimization. This runs under the assumption that recommendations are truthful (non-noised), as well as independently and identically distributed (i.i.d.).

- Semi-Robust matrix completion, which handles the case that users don't necessarily rate all movies randomly. We can't always expect that recommendations are i.i.d. when dealing with real users, due to the range of movie popularities and user interests, and so this algorithm is built to provide accurate recommendations if the input data is non i.i.d.. The semi-robust matrix completion algorithm does not handle cases where the input data is noisy or otherwise non-truthful.

For more details, please see our report for CSCI 2952Q - Robust Algorithms for Machine Learning, where we explore both the standard and semi-robust matrix completion algorithms.

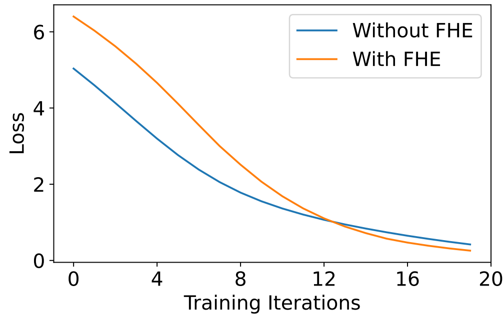
4.3 Miscellaneous Design Choices

Matrix Completion on Request. As mentioned in Section 3.1.4, the server runs matrix completion to provide recommendations to users. Matrix completion needs to be run before the user attempts to access a rating, as we want to guarantee that the user can always get a rating back from the recommendation algorithm when requested. To do this, we run the matrix completion algorithm as the first step when the user asks to receive a rating from the server, as can be seen in Figure 1. Depending on the system's usage, there are better ways to run this to maximize efficiency; we discuss this further in Matrix Completion Call Timing.

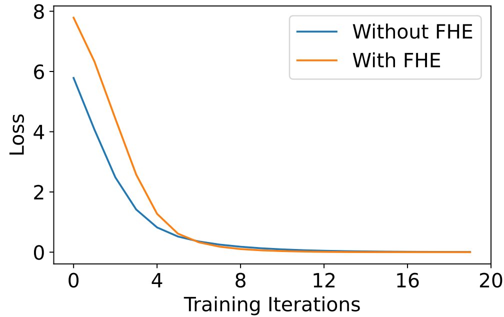
Salting. As mentioned in Section 3.2, when encrypting data to be stored on the server, we use AES symmetric encryption. Our implementation specifically uses zero-salt for this hybrid encryption scheme, but this could alternatively be done via generating different salts k_{iv} and b_{iv} for each map's k and b and store them securely on the server (instead of using zero-salts), and this method could be explored in the future.

Combiner Key Storage. We choose to simplify our threat model such that we trust both the user and the storage the user uses. However, a more comprehensive threat model may choose to exclude the user storage from the set of trusted items. This is why we choose to secure the symmetric encryption key by storing it in server storage. As a result, it would be ideal if the key database that the combiner uses only stores the asymmetric encryption's public keys such as v_k but not the private keys like s_k, which would instead be stored using hybrid encryption (a key encrypt key scheme with symmetric encryption this time instead of asymmetric encryption) on the server. However, in the current implementation, we store both public and private keys in the key database as a simplification.

Alternatively, we can simplify the design of the overall system by choosing to use symmetric encryption instead of hybrid encryption, and never need to store any keys since they can always be recovered by Password Key Derivation within the combiner from a password or IV known only to the combiner.



(a) Loss curves for the secure versus insecure matrix completion algorithms.



(b) Loss curves for the secure versus insecure robust matrix completion algorithms.

Figure 2: Loss curves of secure versus insecure matrix completion on both the robust and non-robust algorithms and a 10x10 user/movie rating matrix with randomly generated ratings. Both systems achieve similar loss curves in this scenario.

5 Evaluation

To evaluate the efficacy and performance of SecRec, we run SecRec on a local machine with 64 GB of RAM, and an 8-core Intel CPU (i7-10870H). We found this to be sufficient as the FHE encryption scheme, and thus SecRec, was most bottlenecked by RAM.

This evaluation aims to answer the following 2 questions:

1. How well does SecRec preserve the accuracy of recommendation algorithms? (Section 5.1)
2. How does SecRec's runtime compare to the equivalent non-secure recommendation algorithms? (Sections 5.2 and 5.3)

5.1 Recommendation Quality

To evaluate the quality of SecRec's generated recommendations, we evaluate its performance on matrix completion relative to non-encrypted implementations. We assess matrix completion performance by using the objective (loss) function of matrix completion. It suffices to measure this function, as it leads to provably optimal completed matrices [2].

SecRec Params	Insecure Time (s)	Secure Time (s)	Runtime Ratio
Non-Robust, 5x5 Matrix	0.0149	457.089	30677.1
Non-Robust, 10x10 Matrix	0.0467	3000.40	64248.4
Non-Robust, 20x20 Matrix	0.1967	20717.6	105325.9
Robust, 5x5 Matrix	0.0223	579.933	26006.0
Robust, 10x10 Matrix	0.05713	3318.22	58081.9
Robust, 20x20 Matrix	0.2638	26440.1	100227.8

Figure 3: Comparison of the runtime of secure versus insecure matrix completion algorithms with varying parameters. Secure matrix completion falls significantly behind the insecure equivalent algorithm, and also scales poorly with increasing size.

Figure 2 shows a comparison of losses from SecRec's matrix completion algorithm in the FHE and non-FHE (plaintext) settings. Although SecRec, when using FHE, converges slower than a non-FHE implementation of matrix completion, it still converges to roughly the same result. The resulting matrices are as optimal as matrices generated from training on unencrypted, plaintext data. SecRec's generated recommendations are accurate, but as the longer time taken to converge would indicate, the main cost of SecRec is in performance.

5.2 Matrix Completion Performance

We evaluate SecRec's performance by comparing its runtime with a non-FHE implementation of matrix completion on matrices of various size. As noted previously, SecRec incurs significant memory overhead from use of FHE schemes. Matrices larger than 20x20 in size incur too much of a memory cost to be reasonably tested using FHE on our machines, so we omit memory performance metrics.

The result of the performance comparison is shown in Figure 3. We see that the cost of performing matrix completion on FHE data is significant, taking more than 30,000 times more time than matrix computation on non-FHE data. This poor performance ratio grows with size of input data, with 20x20 matrices incurring more than 100,000 times more time overhead. Matrix completion requires many matrix multiplications, which grows non-linearly with matrix size. While this is expected due to the current FHE implementations and because we must frequently reset the error of the encryption by decrypting and re-encrypting the matrices, SecRec's time overhead leaves much to be desired. We discuss strategies to mitigate the number of times the matrix completion algorithm must be invoked in Section 6.

Is FHE Enabled?	Is PIR Enabled?	Non-Robust Runtime (s)	Robust Runtime (s)
Yes	Yes	609.20	1051.01
Yes	No	598.83	1028.14
No	Yes	1.2405	1.2977
No	No	0.0872	0.1763

Figure 4: Comparison of the runtime of the end-to-end run when a user requests a rating, using a 10x10 ratings matrix and varying whether FHE or PIR are enabled. FHE results in most of the slowdown, but PIR results in a noticeable performance slowdown of its own.

5.3 End-To-End Performance

Finally, we evaluate the whole system via simulating a user requesting a rating, tested with and without FHE and with and without PIR. For the purposes of this simulation we run all 3 components (user, combiner, and server) in the same program, meaning there is no network or inter-process communication-related latency in these performance numbers. We test these variations of the SecRec system on a 10x10 ratings matrix with a 30% chance of having a randomly generated true rating (otherwise that cell will be filled in during matrix completion). Note that here, a system without FHE but with PIR will store the matrix encrypted, but decrypt before performing matrix completion. Thus, PIR is still performed under encryption, but the actual matrix completion calculation is done unencrypted.

Figure 4 shows the end-to-end performance results. We see that FHE incurs the most time overhead, as PIR adds a time overhead that is significantly less than the overhead FHE causes. This is expected for similar reasons as discussed in Section 5.2, and this result demonstrates that aside from the FHE overhead, SecRec performs reasonably well.

6 Discussion and Future Work

In this paper, we show that SecRec is a working system for running encrypted recommendation algorithms in scenarios where the server is assumed to be untrusted. We describe a system by which FHE can be adapted for a recommendation algorithm setting, and extend the work of Tiptoe [3] by showing that we can still perform secure computations in a use case with more complex algorithms that need to be performed encrypted and a larger portion of the data that is untrusted.

As discussed in Sections 5.2 and 5.3, the performance of SecRec is poor and scales poorly, which is a current limitation of FHE. There are several paths to improving the usability of SecRec which we did not have the time to do, but which would serve as interesting future work to explore. These include the following:

Privacy Budgets for Secure Wrappers. As mentioned in Section 4.1, information can leak to the server via the secure wrappers we provide to the server to offset some functionality that is either not possible or extremely expensive in a FHE scheme. In future versions of SecRec, to reduce the amount that the server can learn about intermediate computation values, we propose that the secure wrappers be equipped with privacy budgets. This budget would look identical to the privacy budgets described in works such as PrivateKube [6] and would limit the number of calculations that the server can use said secure wrappers for such that it can perform the matrix completion without also using it to exploitatively learn about the true data the matrix completion is computing over.

Threshold FHE. As mentioned in Section 3.2, a more secure encryption strategy would leverage threshold FHE. In our current encryption scheme, users of SecRec would need to share the same FHE encryption keys so that the server can perform computations over their data. The server also needs access to the public FHE keys in order to perform computations. Thus, the mechanism used to share FHE keys between users of SecRec is communication with the server.

This method of key sharing is problematic because the server is untrusted and may not share the correct key. One way to solve this problem is switch to a threshold FHE scheme. FHE keys would be generated by and shared between users, then sent to the server. In future versions of SecRec, we could implement threshold FHE by switching our encryption scheme from a FHE scheme to a threshold FHE scheme and implementing necessary thresholds for decryption in the combiner.

Additional FHE matrix operations. Our implementation of SecRec required the wrappers described in Section 4.1 because of the exponentially accumulating error of multiplication on ciphertexts. Rather than using wrappers which involve decrypting the underlying ciphertexts, there exists algorithms to perform approximations of these functions without decryption to still stay in the FHE space [1]. We implemented versions of these algorithms for performing clipping, inversion, and square root operations, but did not include them in the current version of SecRec due to concerns with accumulated error from encrypted multiplication. In future versions of SecRec, we could change the homomorphic encryption implementation we use to better support multiplication and increase error tolerance. We could then also incorporate these algorithms to perform additional operations on FHE data, like normalization of vectors.

Performing SVD under FHE. Our current implementation of singular value decomposition is far more inefficient compared to standard SVD implementations found in libraries like NumPy or SciPy. These libraries make calls to underlying SVD solvers like LaPack or ProPack, which cannot operate correctly on encrypted data. Future iterations of Se-

cRec could help improve performance by implementing more advanced and optimized algorithms for computing SVDs, similar to those used in existing libraries. It is worth noting that these implementations would likely still be slower than existing library implementations since any FHE-compatible SVD implementation must be written natively in languages like Python, whereas LaPack and ProPack are implemented in MatLab, and invoked via bindings in other languages.

Robust matrix completion under FHE. Our semi-robust matrix completion algorithm, as described in Section 4.2, allows for robust computation of recommendations when the inputted data we have access to is not independent and identically distributed. It does not, however, account for the case where users upload noisy or otherwise incorrect ratings (the user misinterprets their own evaluation of a movie). Matrix completion algorithms exist for handling these noisy data cases. These algorithms often generate multiple candidate matrices and select the one with the smallest nuclear norm. However, existing implementations of these algorithms are not compatible with fully homomorphic encryption. In future versions of SecRec, we would like to implement additional robust algorithms on FHE data. For example, implementing a noise-resistant algorithm would require an implementation of an FHE-compatible nuclear norm.

Matrix Completion Call Timing. As mentioned in Matrix Completion on Request, we currently run the matrix completion algorithm as soon as a user requests a rating from the server. If we expect user rating requests to be infrequent, this may be an efficient timing, but in larger-scale systems, we can expect there to be many requests per second, and so it becomes inefficient to always run matrix completion when a user requests a rating.

There are two alternatives to potentially explore for efficiency purposes. The first is that if we instead expect users to infrequently update their ratings in the system, we can call it upon sending a rating to the server. The other is that we can perform a form of periodic update, in which the server periodically updates a user-facing ratings matrix using the matrix completion algorithm, and when the user updates a rating it would be updated in a non-user-facing ratings matrix. This would drastically improve user-facing performance, although there would be a buffer time until their rating updates reflect in the recommendations they see. Both are worth exploring to make this more feasible for use in a real system.

Division of Work

- Siddharth developed the full Recommender API including all components and storage, integrated FHE and implemented hybrid encryption, built the secure wrappers, and built the matrix completion algorithm. He also built storage visualizers for the class demo. Finally, he wrote the Implementation section of the report.
- Ed developed and implemented secure, FHE-compatible subroutines for matrix completion and benchmarking, such as the secure SVD implementation. He also implemented and integrated the FHE-compatible semi-robust matrix completion algorithm. Ed contributed to the sections of this report detailing FHE wrapper functions, benchmarking results, and future work relating to FHE.
- Will developed the PIR scheme and integrated it with the existing FHE scheme, and developed and ran benchmarks for evaluation of the system. He also created all the figures for the paper, and wrote and formatted all but the Implementation section of the paper, as well as revising the Implementation section to better match the flow of the rest of the paper.

References

[1] Jung Hee Cheon, Dongwoo Kim, Duhyeong Kim, Hun Hee Lee, and Keewoo Lee. Numerical method for comparison on homomorphically encrypted numbers. Cryptology ePrint Archive, Paper 2019/417, 2019. https://eprint.iacr.org/2019/417.

[2] Rong Ge, Chi Jin, and Yi Zheng. No spurious local minima in nonconvex low rank problems: A unified geometric analysis. *CoRR*, abs/1704.00708, 2017.

[3] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with tiptoe. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 396–416, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3600006.3613134.

[4] Netflix Inc. Recommendations. https://research.netflix.com/research-area/recommendations.

[5] Google LLC. Ranking results - how google search works. https://www.google.com/search/howsearchworks/how-search-works/ranking-results/.

[6] Tao Luo, Mingen Pan, Pierre Tholoniat, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. Privacy budget scheduling. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 55–74. USENIX Association, July 2021. https://www.usenix.org/conference/osdi21/presentation/luo.