Formally-Verified Private RAGs for LLMs

Dennis Du   Howard Huang   Zachary Ratliff   Roy Rinberg

CS2540 Final Project
June 20, 2025

Abstract

Retrieval-Augmented Generation (RAG) significantly enhances Large Language Models (LLMs), but introduces privacy vulnerabilities when external databases are hosted on untrusted servers. In particular, even if the stored content itself is encrypted, access patterns can inadvertently reveal sensitive information about user queries or document contents. Moreover, in highly sensitive contexts, even the RAG server itself might be considered untrusted, and thus should not gain knowledge of user queries, retrieved documents, or their access patterns. This work formalizes the concept of *Private RAG*, explicitly addressing these critical security concerns. Our approach leverages a Trusted Execution Environment (TEE) to securely conduct semantic searches, producing document indices without leaking query contents. Even if the underlying hardware or cloud provider is compromised, the TEE ensures confidentiality of both queries and retrieved documents. Our primary contribution is the subsequent secure retrieval of documents from untrusted storage using Oblivious RAM (ORAM). Specifically, we utilize Path ORAM to conceal access patterns, guaranteeing that the storage provider learns neither document contents nor document identities. We rigorously define security within a simulation-based framework and present a formally verified (via Dafny) implementation of the core Path ORAM logic, addressing and correcting errors found in previous implementations.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in natural language understanding and generation. However, their effectiveness is often constrained by the static nature of their training data, limiting their access to real-time information or private, domain-specific knowledge bases. Retrieval-Augmented Generation (RAG) systems [LPP+20] address this limitation by integrating external data retrieval into the generation pipeline, allowing LLMs to incorporate relevant, timely context into their responses. This paradigm significantly enhances the accuracy and applicability of LLMs across various knowledge-intensive tasks.

Despite the power of RAG, deploying these systems, particularly when relying on third-party cloud infrastructure for storing and retrieving documents [Rag25, Nuc25], introduces substantial privacy risks. An untrusted hosting server could potentially infer sensitive information, even if the stored data is encrypted. The content itself might be revealing, or more subtly, the patterns of document access could leak confidential information. For example, accessing documents related to specific medical conditions might inadvertently disclose user health data, while patterns correlated with sensitive corporate activities (like layoffs) could expose strategic plans. These vulnerabilities present a significant barrier to adopting RAG for sensitive applications.

This project directly addresses these privacy concerns by formally defining and implementing the notion of a *private RAG* system. Our primary goal is to enable clients to securely utilize an untrusted server for document storage and retrieval within a RAG framework, ensuring the server learns neither (a) the queries issued by clients, (b) the contents of retrieved documents, nor (c) the clients' document access patterns.

To realize this goal, we employ rigorous methodologies from modern cryptography and formal methods. Central to our approach is *Oblivious RAM* (ORAM) [GO96], a cryptographic primitive explicitly designed to obscure data access patterns from untrusted storage providers. We specifically adopt *Path ORAM* [SDS+18], due to its simplicity and acceptable practical performance.

A significant contribution of this project lies in formalizing the security guarantees of such a system within the *simulation-based security* (i.e., real/ideal) paradigm [Lin17]. This approach provides intuitive, composable security definitions, which, to our knowledge, has not been previously applied to formalize RAG security in this specific context. Furthermore, we enhance the reliability and trustworthiness of our solution by developing a formally verified implementation of core Path ORAM functionalities using the Dafny verification language [Lei10]. Thus, our methodology bridges rigorous manual cryptographic analysis with automated verification tools to reason about both security and implementation correctness.

2 Related Work

In this section, we review relevant related work in three areas. First, we survey cryptographic techniques for private document retrieval from untrusted storage, including Oblivious RAM (ORAM) and Private Information Retrieval (PIR). Next, we examine candidate systems and proposals for Private RAG that aim to integrate secure retrieval with language model pipelines. Finally, we discuss the role of Trusted Execution Environments (TEEs) as a mechanism to ensure data confidentiality and execution integrity even on potentially compromised infrastructure.

2.1 Private Document Retrieval

Beyond concerns related to the RAG server learning the contents of user queries or retrieved documents, the general challenge of securely storing and retrieving documents from untrusted storage has been extensively studied. To address privacy issues associated with retrieving documents from untrusted sources, cryptographic techniques designed to conceal data access patterns are typically employed. Oblivious RAM (ORAM) [GO96], which enables a trusted client to perform read and write operations on an untrusted server without revealing which data blocks are accessed or their access sequences, has been extensively studied and widely deployed in practice for this purpose. For example, the Signal protocol uses Oblivious RAM to perform secure contact discovery [Con22]. Numerous ORAM constructions exist, each offering trade-offs among efficiency, storage overhead, and security guarantees [BMP11, CP13, CLP14, GMOT11, GMOT12, LPM$^+$13, DMN11, GM11]. Path ORAM [SDS$^+$18], noted for its simplicity and practical performance, has been effectively applied in recent private RAG proposals such as Compass [ZPZP24], and forms the foundation of our implementation efforts.

An alternative but closely related cryptographic primitive is *Private Information Retrieval* (PIR) [CKGS98]. PIR allows a client to retrieve a specific document from a database stored on one or more servers without revealing to the servers the identity of the requested item. Existing PIR approaches include information-theoretic PIR, which relies on multiple non-colluding servers, and computational PIR (cPIR), which operates with a single server using cryptographic assumptions that often involve homomorphic encryption techniques[ACLS18, ACS18, ORR21]. However, recent research has also revealed extremely efficient PIR schemes that rely only on one-way functions (OWF) [ZPZS24].

The use of Private Information Retrieval (PIR) in RAG arises from similar privacy goals addressed by ORAM—preventing servers from learning document relevance to user queries. Existing PIR schemes, especially single-server computational PIR (cPIR), naturally support private reads but typically lack efficient support for write operations or dynamic updates. Furthermore, single-server cPIR schemes often incur considerable computational overhead due to reliance on expensive public-key operations like homomorphic encryption, potentially impacting latency and costs. Therefore, while PIR is promising for read-heavy use cases, ORAM remains preferable for comprehensive read-write scenarios, motivating our current focus on Path ORAM.

2.2 Private RAG Systems

Recent work has examined privacy vulnerabilities in RAG systems, most notably the Compass project by Zhu et al. [ZPZP24]. Compass applies Path ORAM to protect the semantic search process, enabling clients to identify relevant document embeddings while keeping the server oblivious to the access pattern. This reduces reliance on the storage server's trustworthiness, but it requires each client to maintain a position map and track the exact location of every document in untrusted storage. While this model works for scenarios where users upload and access only their own documents, it limits the ability to support shared or public data. Incorporating such data would require either constant re-synchronization of ORAM state across users or redundant storage across clients—both of which are impractical at scale.

An alternative privacy-enhancing design is given by Zhou et al. [ZFY25], which focuses on content-level confidentiality. Their scheme encrypts both textual content and vector embeddings before storage and organizes ciphertexts with a user-isolated key hierarchy, preventing cross-tenant leakage and prompt-injection attacks. However, because the storage server can still observe which ciphertexts are accessed, it does not hide access patterns or query timing.[1]

A third approach, proposed by Cheng et al. [CZW$^+$24], focuses on protecting query privacy for cloud-hosted RAG services rather than addressing access-pattern leakage. Their system, RemoteRAG, first perturbs the user's embedding with Laplace noise to achieve an (n, ϵ)-*DistanceDP* guarantee[2] before sending the noisy vector to the cloud. Subsequently, a partially homomorphic encryption scheme is employed to securely identify and retrieve the top-k relevant documents corresponding to this noisy embedding. While this ensures that the server cannot directly learn the exact user query or its original embedding, it still reveals access patterns through observable ciphertext retrievals or the selected candidate set. Furthermore, the introduction of noise inherently creates a trade-off between retrieval accuracy and privacy.

In contrast to these approaches, our system combines

[1]We note that the work by Zhou et al. [ZFY25] was released in late March 2025, underscoring the timeliness and active development in this area of research.

[2]This privacy guarantee can be viewed as a variant of differential privacy.

semantic search executed within a Trusted Execution Environment (TEE, described in the next section) with Path ORAM-based document retrieval. This design ensures that neither the exact query nor the subsequent document retrieval patterns are revealed to the untrusted storage server. Unlike Compass, our method supports scalable and flexible management of shared or public documents by offloading semantic search securely to the TEE, thus avoiding per-user state synchronization issues. Furthermore, compared to the Privacy-Aware RAG and RemoteRAG approaches, our system explicitly addresses access-pattern privacy, providing stronger, cryptographic security guarantees without compromising retrieval accuracy through noise injection.

2.3 Trusted Execution Environments

Trusted Execution Environments (TEEs) are hardware-supported isolation mechanisms that enable code to execute in protected regions of memory, shielded even from privileged software such as the operating system or hypervisor [SMS+22, LZY+23].A TEE is instantiated via a verifiable boot process, where a trusted processor measures the enclave's initial state and cryptographically attests that the software was loaded correctly. This attestation can be verified by remote parties before provisioning secrets or data. During runtime, the enclave maintains strong isolation enforced through access controls and memory encryption, and may also support secure persistent storage through "sealing" and trusted I/O paths.

TEEs operate under a threat model that assumes the secure processor is trusted, while treating the OS, hypervisor, and external memory as potentially malicious. They defend against software-based attacks and many hardware snooping threats, but typically do not address physical tampering or side-channel leakage without additional mitigations. TEEs are widely used in secure cloud computing settings since they allow the cloud to perform sensitive operations without learning anything about the user's inputs, intermediate state, or outputs.

3 Threat Model

In our system, we consider a cloud-hosted RAG service, where the cloud provider offers both hardware equipped with Trusted Execution Environments (TEEs, §2.3) and external storage for documents. The RAG service conducts semantic search and document retrieval on behalf of clients. Specifically, the semantic search (identifying document indices most relevant to a user query) is securely executed within a TEE. However, the actual documents are stored separately in an untrusted storage environment provided by the same cloud provider.

Our threat model explicitly assumes that the cloud provider, storage server, and network are malicious or

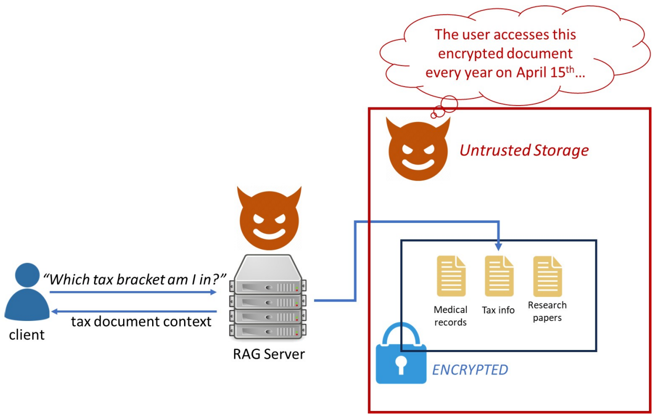


Figure 1: Even if document contents are encrypted, an untrusted server may infer sensitive information by observing access patterns. For example, a client who retrieves the same document every April 15th (tax day in the United States) may inadvertently reveal the nature of the document through timing and frequency alone.

potentially compromised. We only trust the TEE itself as a secure computing environment, which includes encrypted access to secure (trusted) working memory, ensuring the confidentiality and integrity of the operations performed within the enclave. The primary adversarial goal is to infer sensitive information about user queries or retrieved document identities by analyzing access patterns.

4 Formalizing Private RAG

In this section, we formalize security definitions for private RAG. We start by introducing the notion of a RAG scheme. Throughout our definitions, we consider all storage as being represented as a flat array of equally sized blocks.

Private RAG Scheme. A *Private Retrieval-Augmented Generation (RAG) scheme* Σ is a tuple of probabilistic polynomial-time (PPT) algorithms $\Sigma = (\mathsf{Setup}, \mathsf{Query})$ defined as follows:

- $\mathsf{Setup}(\lambda, D, B)$: Initializes a flat array O to store $N = |D|$ documents, each of size B bits. The input set D consists of documents of size at most B; shorter documents are padded with null bytes to reach uniform length. The parameter λ denotes the security parameter.

- $\mathsf{Query}(q)$: On input a user query q, deterministically maps q to an identifier i, retrieves $O[i]$, and outputs the corresponding document d (or $\perp$ if i is undefined).

While the generic RAG scheme described above simply maps user queries to relevant documents, ensuring privacy in this process has motivated a number of recent system designs (§2).

To reason about security in this setting, we use the simulation-based paradigm from modern cryptography, in which we define two experiments: one capturing the behavior of a real world protocol and the other describing an ideal system that reveals only minimal information. Security is then established by showing that no efficient adversary can distinguish between the two.

Simulation-based Security. Let $\mathcal{E}$ denote the environment, $\mathcal{A}$ the adversary that observes the access patterns of untrusted storage O, and $\mathcal{S}$ a simulator. Define the two experiments below for a security parameter λ.

Algorithm 1 Real World Experiment

```
$\underline{\textsc{Real}^{\Sigma}_{\mathcal{E},\mathcal{A}}}(\lambda)$
1: $(B, D, m) \leftarrow \mathcal{E}(1^\lambda)$
2: $\mathcal{E}$ runs $\Sigma.\mathsf{Setup}(\lambda, B, D)$
3: for $k = 1$ to $m$ do
4:     $q_k \leftarrow \mathcal{E}(1^\lambda)$
5:     $\mathcal{E}$ issues $\Sigma.\mathsf{Query}(q_k)$
6: end for
7: The full access pattern of $O$ and the query sequence
   $(q_1, \ldots, q_m)$ are given to $\mathcal{A}$
8: $v \leftarrow \mathcal{A}(1^\lambda)$
9: $b \leftarrow \mathcal{E}(1^\lambda, v)$
10: return $b$
```

Algorithm 2 Ideal World Experiment

```
$\underline{\textsc{Ideal}^{\mathcal{F}}_{\mathcal{E},\mathcal{S}}}(\lambda)$
1: $(B, D, m) \leftarrow \mathcal{E}(1^\lambda)$
2: $\mathcal{E}$ runs $\mathcal{F}.\mathsf{Setup}(\lambda, B, D)$ where $\mathcal{F}$ is the ideal func-
   tionality that operates on hidden storage $H$
3: for $k = 1$ to $m$ do
4:     $q_k \leftarrow \mathcal{E}(1^\lambda)$
5:     $\mathcal{E}$ issues $\mathcal{F}.\mathsf{Query}(q_k)$
6: end for
7: The query transcript $(q_1, \ldots, q_m)$ is given to $\mathcal{S}$
8: $v \leftarrow \mathcal{S}(1^\lambda)$
9: $b \leftarrow \mathcal{E}(1^\lambda, v)$
10: return $b$
```

x

Definition 1 (Secure Private RAG)**.** *A Private RAG scheme* Σ *is* secure *if for every PPT adversary* $\mathcal{A}$ *there exists a PPT simulator* $\mathcal{S}$ *such that for all PPT environments* $\mathcal{E}$*,*

$$\left|\Pr\left[\textsc{Real}^{\Sigma}_{\mathcal{E},\mathcal{A}}(\lambda) = 1\right] - \Pr\left[\textsc{Ideal}^{\mathcal{F}}_{\mathcal{E},\mathcal{S}}(\lambda) = 1\right]\right| \leq \mathit{negl}(\lambda),$$

where negl is a negligible function in λ*.*

In other words, anything that the adversary can learn by observing the full access patterns in the real world — is equivalent to what a simulator in the ideal world could learn, even though the simulator observes no memory accesses at all, as all reads and writes are handled through a trusted third party (hidden storage).

It turns out that ORAM suffices to achieve Private RAG security. Recall the security definition for ORAM.

Definition 2 (ORAM Security [SSS11])**.** *Let*

$$y = \big((\mathsf{op}_M, a_M, \mathsf{data}_M), \ldots, (\mathsf{op}_1, a_1, \mathsf{data}_1)\big)$$

denote a sequence of M *data requests issued by the client, ordered from most recent (index* 1*) to oldest (index* M*). Each triple* $(\mathsf{op}_i, a_i, \mathsf{data}_i)$ *is either a* read *operation* $(\mathsf{read}, a_i, \perp)$ *or a* write *operation* $(\mathsf{write}, a_i, \mathsf{data}_i)$*, where* a_i *is the identifier of the logical block accessed and* data_i *the data written (empty* $\perp$ *for reads).*

Let $A(y)$ *be the (possibly randomized) sequence of* physical *accesses carried out on the remote storage when the logical request sequence* y *is executed. An Oblivious RAM (ORAM) construction is* secure *if it satisfies both of the following properties:*

(1) ***Access-pattern indistinguishability.*** *For any two request sequences* y *and* z *of equal length, the distributions* $A(y)$ *and* $A(z)$ *are computationally indistinguishable to every polynomial-time distinguisher that does not possess the client's secret state.*

(2) ***Correctness.*** *For every request sequence* y*, the client receives responses that are consistent with sequentially executing* y *except with negligible probability* $\mathrm{negl}(\lambda)$*, where* λ *is the security parameter.*

Given a secure ORAM construction, we can build a secure Private RAG scheme by ensuring that all accesses to the untrusted storage O (including both reads and writes) are mediated by the ORAM protocol.

Lemma 1 (ORAM suffices for Secure Private RAG)**.** *Let* Σ *be a Private RAG scheme whose* Query *algorithm accesses the untrusted array* O *exclusively through a secure ORAM protocol. Then* Σ *is a secure private RAG (Definition 1).*

Proof. Fix any PPT adversary $\mathcal{A}$ and environment $\mathcal{E}$. In the real experiment $\textsc{Real}^{\Sigma}_{\mathcal{E},\mathcal{A}}$ the view of $\mathcal{A}$ is derived from the distribution of physical access patterns Q_{real} produced by ORAM when the logical query sequence $q_1, \ldots, q_m$ is executed.

In the ideal experiment $\textsc{Ideal}^{\mathcal{F}}_{\mathcal{E},\mathcal{S}}$, the simulator $\mathcal{S}$ learns only the length m of that sequence. To simulate $\mathcal{A}$'s view, $\mathcal{S}$ outputs any fixed ORAM access pattern of length m (e.g., m dummy reads to a fixed address). Let $Q_{\mathcal{S}}$ be this distribution of physical access patterns on O. Then S gives the full access patterns $Q_{\mathcal{S}}$ and queries $(q_1, \ldots, q_m)$ to the algorithm $\mathcal{A}$ and outputs whatever $\mathcal{A}$

outputs. Because ORAM security guarantees that physical patterns for any two logical sequences of equal length are computationally indistinguishable, then $Q_{\mathcal{S}}$ and $Q_{\mathcal{A}}$ are computationally indistinguishable. Thus, in the real-world, the view v output by $\mathcal{A}$ and the view v output by $\mathcal{S}$ are computationally-indistinguishable. Hence

$$\left|\Pr[\text{REAL}^{\Sigma}_{\mathcal{E},\mathcal{A}}(\lambda) = 1] - \Pr[\text{IDEAL}^{\mathcal{F}}_{\mathcal{E},\mathcal{S}}(\lambda) = 1]\right| \leq \text{negl}(\lambda).$$

□

Note that the above security proof captures a relaxation of the informal security properties discussed in (§1). Specifically, the proof assumes that the scheme Σ and the machine executing its code are trusted. Moreover, it assumes the queries issued by the environment $\mathcal{E}$ are revealed to the adversary (and similarly to the simulator). Thus, this security definition captures precisely the guarantees provided by systems such as Compass [ZPZP24], but does not fully reflect scenarios where the execution environment itself is untrusted. Extending these security definitions to explicitly model an untrusted execution environment, and then proving security when the RAG is executed within a TEE (coupled with a standard CCA-secure encryption scheme), is straightforward and omitted here for brevity.

5 Formally-Verified Path ORAM

The problem setup is the following: there is an untrusted server that holds an encrypted database of sensitive information, and a trusted client that wants to read from and write to the database on the server without revealing the memory access patterns to the server. The basic atomic unit of data that the Path ORAM protocol manipulates is called a *block*, each of which has some fixed size of B bits and has an associated address. We represent a block as a pair `(a,data)`, where `a` is the block's address and `data` is the block's contents.

On the server, data is stored in *buckets*. Each bucket has the capacity to hold Z blocks, and in fact always does hold Z blocks: if there is extra space, it is filled with dummy blocks. The client thinks of these buckets as arranged in a binary tree (though the server can just store them as a flat array). The function `ReadBucket(i)` returns all the blocks stored in bucket `i` of the database and decrypts them. The function `WriteBucket(i, blocks)` encrypts and writes `blocks` to bucket `i`, adding encrypted dummy blocks if necessary.

Let N be the number of blocks we wish to store. Then the server's binary tree of buckets (where each bucket is a node) should be of height $L = \lceil \log_2(N) \rceil - 1$ and contain 2^L leaves. (Notice that means we have approximately N leaves and approximately $2N$ total buckets in the tree.) The levels are numbered from 0 to L, with 0 being the root and L being the leaves. The leaves are numbered from 0 to $2^L - 1$. There is a unique path from the root to any leaf x; let $\mathcal{P}(x)$ denote that path. Let $\mathcal{P}(x, \ell)$ be the bucket at tree level ℓ in that path.

The client needs to store a *stash* S and a *position map* `position`, whose values persist from one call of `Access` to the next. (That is, each call to `Access` uses the previous values of S and `position` to inform how it runs, and then modifies them in preparation for future calls to `Access`.) The stash is a local cache of blocks that the client holds. The position map is a map indicating, for each block address, which leaf bucket that block is mapped to; `position[a] ==` x indicates that the block with address `a` is currently in some bucket on the path $\mathcal{P}(x)$ or is in the stash S. The stash starts out empty and the position map starts with random values from DiscreteUniform$(0, 2^L - 1)$.

Directly using `ReadBucket` and `WriteBucket` to access the server's database would reveal memory access patterns, so instead, all database accesses are performed using `Access(op, a, data*)`. To read the block with a given address, we set `op = read` and `a` to be the address: the function then returns the contents of the block with the specified address. To write to the block with a given address, we set `op = write`, `a` to be the address, and `data*` to be the new data to write to that block: the function then updates the specified block with the new data and returns the old contents of the block.

Algorithm 3 `Access(op, a, data*)`

```
1:  $x \leftarrow$ position[a]
2:  position[a] $\leftarrow$ DiscreteUniform$(0, 2^L - 1)$
3:  for $\ell \in \{0, \dots, L\}$ do
4:      $S \leftarrow S \cup$ ReadBucket$(\mathcal{P}(x, \ell))$
5:  end for
6:  data $\leftarrow$ The data portion of the block (a,data) in $S$ with address a
7:  if op = write then
8:      $S \leftarrow (S - \{$(a,data)$\}) \cup \{$(a,data*)$\}$
9:  end if

10: for $\ell \in \{L, \dots, 0\}$ do
11:     $S' \leftarrow \{$(a',data') $\in S : \mathcal{P}(x, \ell) = \mathcal{P}($position[a']$, \ell)\}$
12:     $S' \leftarrow$ Select $\min(|S'|, Z)$ blocks from $S'$
13:     $S \leftarrow S - S'$
14:     WriteBucket$(\mathcal{P}(x, \ell), S')$
15: end for

16: return data
```

Lines 1-2 get the leaf bucket x that the given block with address `a` currently maps to and then remaps the block to a random bucket for the future. Lines 3-5 read all the buckets from server along the path from the root to x and stores their blocks in the stash S. Lines 6-9 reads data from and writes data to the requested block (which is now guaranteed to be in the stash). Lines 10-15 write the path back to the server (and also possibly writes other blocks that are still in the stash if they are allowed to go in those buckets), filling buckets from leaf to root. A block can only be placed in a bucket if it is mapped (by the position map) to a descendant of that bucket.

5.1 Dafny Implementation

We implemented the Path ORAM protocol in Dafny with the goal of formally verifying its correctness: that is, we aim to verify that from the client's perspective, the `Access` function behaves as one would expect, with reads and writes having the same effect as they normally would without Path ORAM, and that the invariants of Path ORAM are adhered to. The implementation can be found at Private-RAG Github.

Each block is represented by the algebraic data type `datatype Block = Block(addr: int, data: string)`, indicating the address and contents of a block. Each real block has a unique address from 0 to $N-1$, and an address of -1 represents a dummy block that is used solely to fill up an otherwise incomplete bucket. The class `Database` represents the database on the untrusted server and consists of an object of type `seq<seq<Block>>`. Each inner sequence has length Z and represents Z blocks collected together in a bucket, and the outer sequence that contains all these buckets represents a key-value database, associating indices with buckets. From the Path ORAM perspective this is the flattened tree of buckets. The database is unaware of what happens inside each bucket; reads and writes happen at the bucket level. So the `Database` class has a function `ReadBucket`, which takes in the index of a bucket in the database and outputs the bucket stored there as a `seq<Block>` object. It also has a function `WriteBucket`, which takes in an index of the database and the contents of a bucket as a `seq<Block>` and writes the bucket to the specified index. We do not perform encryption of the blocks in our Dafny implementation since this is unnecessary for verifying correctness, but ultimately the encryption would ensure that the database only sees bucket accesses—the contents of the buckets themselves would be incomprehensible.

We then create a class `PathORAM` which defines the Path ORAM protocol, and in particular the function `Access`, which implements Algorithm 3. The most important attributes are `tree`, which represents the database in the server; `posMap`, which represents the position map and is stored as an array mapping block addresses to leaf bucket indices; and `stash`, which represents the stash and is stored as a map whose keys are the block addresses and values are the block contents. The class also stores the total number of blocks to store N, the number of blocks in each bucket Z, and some derived values. We also have a recursively defined function `PathNodes` which returns the path $\mathcal{P}(x)$ from the root to a given leaf x; starting from any leaf x, we compute $(x-1)/2$ to get its parent, and recursively call the function to get all the ancestors going up to the root. We are using the flat-array representation of binary trees in which index 0 represents the root and the two children of the node at any index i are located at indices $2i+1$ and $2i+2$. We sample a random new value for the position map using `RandomLeaf`, which uses a non-deterministic choice to pick a value between 0 and 2^L-1. Although this does not guarantee uniform randomness, it suffices for a proof of correctness.

We had used an existing Python implementation of Path ORAM as a starting point, but we found that it had multiple errors. For example, we noticed the following in the Python implementation:

- The concepts of blocks and buckets were conflated, so that each bucket could store only one block.
- The number of leaves was incorrectly calculated.
- The `Access` function failed to return the contents of the requested block, so a read had no effect from the client's perspective.
- Writing blocks from the stash to the server was done in a way that did not correspond with lines 10-15 of Algorithm 3.

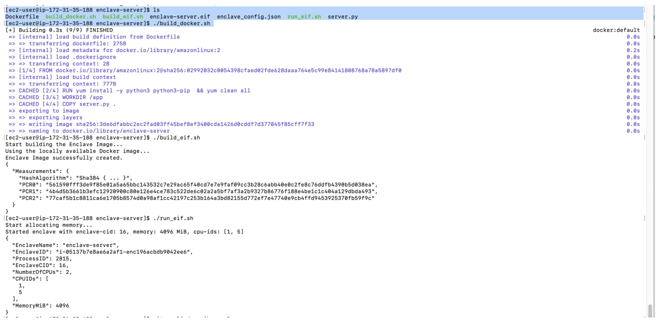
We believe our corrected implementation can serve as a contribution in its own right.

Verifying with Dafny ended up being tricker and more time-consuming than expected. Since the data structures and the access algorithm were quite complicated, Dafny would persistently be unable to verify the code would not crash. For example, it would complain that array indices could not be proved to be in the valid range or that sets we are trying to take elements from could not be proved to contain any elements. A lot of time was spent identifying assertions, loop invariants, and lemmas that would convince Dafny that there would be no runtime errors of this form. The issue is that it is usually not immediately clear what Dafny does not understand and what holes need to be filled in order for it to identify the proof. As a result, the work mostly becomes trial and error, adding a large number of assertions and loop invariants and observing what Dafny can and cannot verify in order to pinpoint the gaps in logic, and then repeatedly trying to fill in those gaps in logic with more assertions and invariants. Occasionally certain statements, such as facts from set theory, had to be abstracted out into lemmas. In the end we were able to verify that the code has no memory errors or other similar runtime errors.

6 Implementing Private RAG

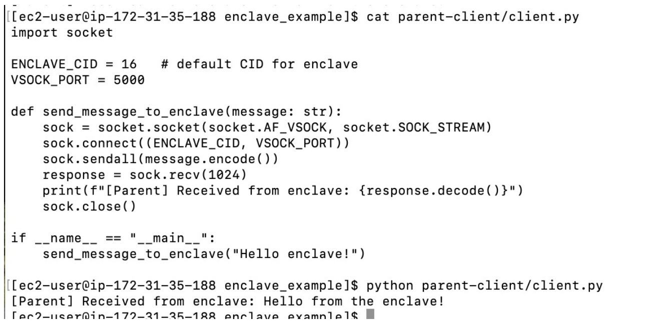
6.1 Outline of RAG Implementation

In our implementation, we use FAISS (Facebook AI Similarity Search) widely adopted due to its efficiency and



```
[ec2-user@ip-172-31-35-188 enclave-server]$ ls
Dockerfile  build_docker.sh  build_eif.sh  enclave-server.eif  enclave_config.json  run_eif.sh  server.py
[ec2-user@ip-172-31-35-188 enclave-server]$ ./build_docker.sh
[+] Building 0.3s (9/9) FINISHED                                                                                   docker:default
 => [internal] load build definition from Dockerfile                                                                         0.0s
 => => transferring dockerfile: 275B                                                                                         0.0s
 => [internal] load metadata for docker.io/library/amazonlinux:2                                                             0.2s
 => [internal] load .dockerignore                                                                                            0.0s
 => => transferring context: 2B                                                                                              0.0s
 => [1/4] FROM docker.io/library/amazonlinux:2@sha256:02992032c0054398cfaed02fde628daaa764e5c99e84141808768a78a5897df0      0.0s
 => [internal] load build context                                                                                            0.0s
 => => transferring context: 777B                                                                                            0.0s
 => CACHED [2/4] RUN yum install -y python3 python3-pip  && yum clean all                                                    0.0s
 => CACHED [3/4] WORKDIR /app                                                                                                0.0s
 => CACHED [4/4] COPY server.py .                                                                                            0.0s
 => exporting to image                                                                                                       0.0s
 => => exporting layers                                                                                                      0.0s
 => => writing image sha256:3de6dfabbc2ec2fad03ff45bef8ef3400cda1426d0cddf7d377045f85cff7f33                                 0.0s
 => => naming to docker.io/library/enclave-server                                                                            0.0s
[ec2-user@ip-172-31-35-188 enclave-server]$ ./build_eif.sh
Start building the Enclave Image...
Using the locally available Docker image...
Enclave Image successfully created.
{
  "Measurements": {
    "HashAlgorithm": "Sha384 { ... }",
    "PCR0": "561590fff3de9f85e01a5a65bbc143532c7e29ac65f40cd7e7e9faf09cc3b28c6abb40e0c2fe8c76ddfb4390b5d038ea",
    "PCR1": "4b4d5b3661b3efc12920900c80e126e4ce783c522de6c02a2a5bf7af3a2b9327b86776f188e4be1c1c404a129dbda493",
    "PCR2": "77caf5b1c8811ca6e1705b8574d0a98af1cc42197c253b164a3bd82155d772ef7e47740e9cb4ffd9453925370fb59f9c"
  }
}
[ec2-user@ip-172-31-35-188 enclave-server]$ ./run_eif.sh
Start allocating memory...
Started enclave with enclave-cid: 16, memory: 4096 MiB, cpu-ids: [1, 5]
{
  "EnclaveName": "enclave-server",
  "EnclaveID": "i-05137b7e8ae6a2af1-enc196acbdb9042ee6",
  "ProcessID": 2815,
  "EnclaveCID": 16,
  "NumberOfCPUs": 2,
  "CPUIDs": [
    1,
    5
  ],
  "MemoryMiB": 4096
}
```

Figure 2: Building a docker image and nitro image, and then running the nitro image inside of a TEE.



```
[ec2-user@ip-172-31-35-188 enclave_example]$ cat parent-client/client.py
import socket

ENCLAVE_CID = 16    # default CID for enclave
VSOCK_PORT = 5000

def send_message_to_enclave(message: str):
    sock = socket.socket(socket.AF_VSOCK, socket.SOCK_STREAM)
    sock.connect((ENCLAVE_CID, VSOCK_PORT))
    sock.sendall(message.encode())
    response = sock.recv(1024)
    print(f"[Parent] Received from enclave: {response.decode()}")
    sock.close()

if __name__ == "__main__":
    send_message_to_enclave("Hello enclave!")

[ec2-user@ip-172-31-35-188 enclave_example]$ python parent-client/client.py
[Parent] Received from enclave: Hello from the enclave!
[ec2-user@ip-172-31-35-188 enclave_example]$
```

Figure 3: Running a client to interface with the server inside of the TEE, from outside of the TEE

scalability [DGD+25], specifically embedding the majority of Wikipedia into a RAG system.[3] As far as we can tell, despite many academic papers citing building a RAG on wikipedia, this is the first open-source wikipedia-based RAG system. [4]

Our goal in this project is to build a RAG system within a TEE that uses PathORAM. We first outline how one would build a RAG, with the specific use-case of building a RAG using Wikipedia pages. We provide extensive documentation of what we do for this in the README provided in the github repo: Wiki-RAG Github. We keep this section short in the main body, and provide an in-depth description of how to build, run, and interface with the RAG system in the appendix A.

6.2 Running in a TEE

Figures 2 and 3 illustrate the workflow of setting up and interacting with a Retrieval-Augmented Generation (RAG) model deployed securely within a Trusted Execution Environment (TEE) using AWS Nitro Enclaves.

In Figure 2, we show a screenshot of the commands associated with creating a Docker image that encapsulates the RAG model and its serving infrastructure. This Docker image is then converted into a Nitro Enclave Image File (EIF), which is specifically formatted to run securely within the AWS Nitro TEE. Finally, we show code for initiating the AWS Nitro enclave.

Figure 3 is a screenshot of the client application outside the TEE. The client communicates with the secure server running within the Nitro enclave through a virtual socket (vsock).

We do note that we were unable to get the FAISS-based RAG to run within the TEE, due to dependency issues. In short, our RAG implementation was initially developed and thoroughly tested with Python 3.10. Although functional on Python 3.9 in environments such as macOS and ubuntu docker images, the libraries we built our RAG code with were not supported on the Amazon Linux Docker image environment we needed to use for developing our TEE code. To provide more details, we found the following compatibility issues:

- **Docker Base Image Constraints:** AWS Nitro Enclaves require the use of the Amazon Linux Docker image, specifically `amazonlinux:2`.
- **Python Version Compatibility:** The Amazon Linux 2 image does not officially support Python 3.10, limiting compatibility to Python 3.9 or earlier.
- **Dependency Compatibility Issues:** Critical dependencies for our RAG system, such as `faiss_cpu` and `langchain`, exhibited compatibility problems when installed on the `amazonlinux:2` image with Python 3.9. Specifically, we found that `faiss_cpu` does not provide pre-built binaries fully compatible with Amazon Linux, and we were unable to build *faiss_cpu* from scratch.

6.3 PathORAM implementations with Wikipedia

To ensure the pattern of RAG accesses to the database of Wikipedia articles remains private, we implemented a Path ORAM system in Python that serves Wikipedia articles to a client using Path ORAM accesses. We downloaded a subset of Wikipedia articles to load into Path ORAM. This Python implementation along with the subset of Wikipedia articles is available at Private-RAG Github; the README explains how to run it.

In this implementation, the bucket size is set to $Z = 4$ blocks and the block size is set to $B = 4096$ bytes. Articles that are longer than 4096 bytes are divided among multiple blocks, with 4096 bytes of each article filling each block, except the last block for each article, which contains the remaining article data and is then padded with spaces to reach 4096 bytes. The block size is chosen to maintain a balance between access time overhead and storage space overhead: Larger block sizes reduce the number of accesses that need to be made per article

[3]We take the top 100,000, 1,000,000, and 10,000,000 english articles by page-view.

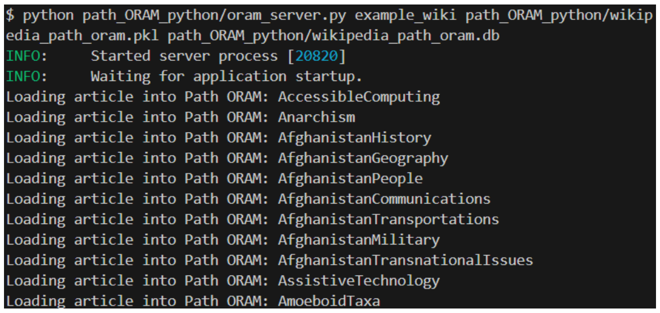
[4]While not a contribution directly to the Formal Methods community, it still seems to be a meaningful contribution to the open-source development.

since articles will be divided among fewer blocks, while smaller block sizes reduce the amount of padding needed to reach the desired block size. In total, our subset of Wikipedia requires $N = 2970$ blocks, and each one is given an address from 0 to $N - 1$. A dictionary is maintained mapping article titles to the list of the addresses of all the blocks storing that article's contents. This allows conversion between an article title and block addresses, which are what the Path ORAM `Access` function understands. Even though Path ORAM will move the blocks around the database at random, the address of each block never changes. That is, the address does not say where the block is located, but is rather an identifier for the block, just like an article title.

When the server is run for the first time, a SQL database is set up with the appropriate number of buckets according to the Path ORAM protocol, which is 4095 buckets. (The tree height is $L = \lceil \log_2 N \rceil - 1 = 11$, resulting in $2^L = 2048$ leaf nodes and 4095 nodes in total.) This SQL database represents the untrusted storage. As such, it does not understand anything about the structure we are talking about here, including the fact that it represents a tree; the database itself is simply a key-value store with keys as integers and values as strings. From the Path ORAM perspective though, the keys are bucket indices with the indices labeling the nodes of a tree, and the values are JSON-serialized buckets, which are lists of blocks, with each block being a two-element list consisting of its address and contents. Initially the database buckets are completely filled with dummy blocks, which have address -1 and whose contents are a string of 4096 spaces. The blocks holding the Wikipedia articles are loaded one-by-one into the database using the Path ORAM `Access` function in write mode.

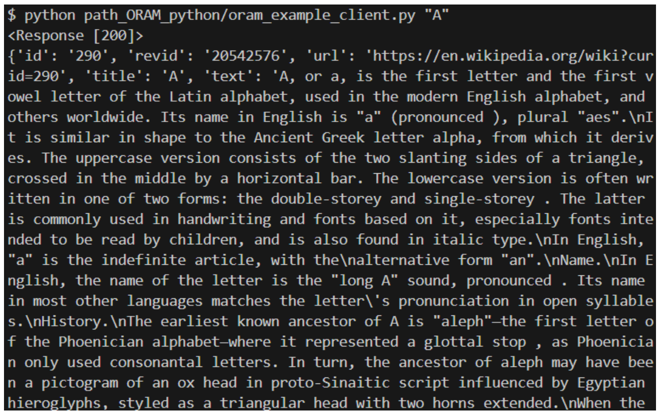
The original Wikipedia articles take 9.8 MB of storage. The SQL database for Path ORAM that stores these articles takes 66 MB of storage, which is larger by a factor of about 6.7. This is to be expected, since Path ORAM calls for about as many buckets in the database as there are blocks to store, even though each bucket has the space to hold $Z = 4$ blocks and mandatorily holds that many blocks (with dummy blocks to pad). This already contributes a factor of 4. The other major source of storage overhead is the padding bytes needed within the blocks to reach 4096 bytes. This overhead is necessary to ensure the database remains oblivious to the access patterns.

The state of the Path ORAM object is saved in a pickle file after every operation so that when the server program is terminated and later restarted, it can simply pick up where it left off. Crucially, the pickle contains Path ORAM's stash and position map, which cannot be lost—the stash holds the blocks that have not been written back to the database, and the position map tells us where to find each block. Specific to the Wikipedia application, it also needs to hold the mapping from titles to



```
$ python path_ORAM_python/oram_server.py example_wiki path_ORAM_python/wikip
edia_path_oram.pkl path_ORAM_python/wikipedia_path_oram.db
INFO:     Started server process [20820]
INFO:     Waiting for application startup.
Loading article into Path ORAM: AccessibleComputing
Loading article into Path ORAM: Anarchism
Loading article into Path ORAM: AfghanistanHistory
Loading article into Path ORAM: AfghanistanGeography
Loading article into Path ORAM: AfghanistanPeople
Loading article into Path ORAM: AfghanistanCommunications
Loading article into Path ORAM: AfghanistanTransportations
Loading article into Path ORAM: AfghanistanMilitary
Loading article into Path ORAM: AfghanistanTransnationalIssues
Loading article into Path ORAM: AssistiveTechnology
Loading article into Path ORAM: AmoeboidTaxa
```

Figure 4: Loading Wikipedia articles into Path ORAM during server initialization.



```
$ python path_ORAM_python/oram_example_client.py "A"
<Response [200]>
{'id': '290', 'revid': '20542576', 'url': 'https://en.wikipedia.org/wiki?cur
id=290', 'title': 'A', 'text': 'A, or a, is the first letter and the first v
owel letter of the Latin alphabet, used in the modern English alphabet, and
others worldwide. Its name in English is "a" (pronounced ), plural "aes".\nI
t is similar in shape to the Ancient Greek letter alpha, from which it deriv
es. The uppercase version consists of the two slanting sides of a triangle,
crossed in the middle by a horizontal bar. The lowercase version is often wr
itten in one of two forms: the double-storey and single-storey . The latter
is commonly used in handwriting and fonts based on it, especially fonts inte
nded to be read by children, and is also found in italic type.\nIn English,
"a" is the indefinite article, with the\nalternative form "an".\nName.\nIn E
nglish, the name of the letter is the "long A" sound, pronounced . Its name
in most other languages matches the letter\'s pronunciation in open syllable
s.\nHistory.\nThe earliest known ancestor of A is "aleph"—the first letter o
f the Phoenician alphabet—where it represented a glottal stop , as Phoenicia
n only used consonantal letters. In turn, the ancestor of aleph may have bee
n a pictogram of an ox head in proto-Sinaitic script influenced by Egyptian
hieroglyphs, styled as a triangular head with two horns extended.\nWhen the
```

Figure 5: Reading the Wikipedia article about the letter A using a query from the client.

block addresses. If the pickle is deleted, since data is lost and it is no longer possible to access the database, the server will do a hard reset by going back to the original Wikipedia articles and loading them into a fresh Path ORAM database.

We also implemented a simple client to make queries to the server. Since we did not connect this portion to a RAG, the user has to make queries that are exact matches for Wikipedia article titles in the database, and the raw contents of the article are returned.

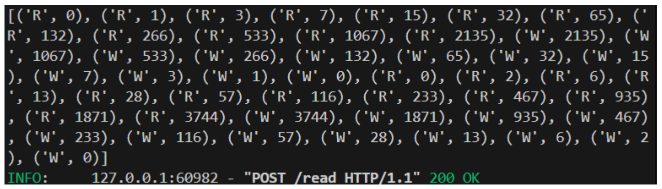
Examples of this Wikipedia for Path ORAM system running are shown in Figures 4, 5, and 6.

6.4 Incomplete Implementations

Our proposed system comprises three core components:

1. A functioning Retrieval-Augmented Generation (RAG) system,
2. A secure RAG deployment within a Trusted Execution Environment (TEE),
3. An operational PathORAM implementation.

In this project, we successfully implemented each of these components individually. However, due to several



```
[('R', 0), ('R', 1), ('R', 3), ('R', 7), ('R', 15), ('R', 32), ('R', 65), ('
R', 132), ('R', 266), ('R', 533), ('R', 1067), ('R', 2135), ('W', 2135), ('W
', 1067), ('W', 533), ('W', 266), ('W', 132), ('W', 65), ('W', 32), ('W', 15
), ('W', 7), ('W', 3), ('W', 1), ('W', 0), ('R', 0), ('R', 2), ('R', 6), ('R
', 13), ('R', 28), ('R', 57), ('R', 116), ('R', 233), ('R', 467), ('R', 935)
, ('R', 1871), ('R', 3744), ('W', 3744), ('W', 1871), ('W', 935), ('W', 467)
, ('W', 233), ('W', 116), ('W', 57), ('W', 28), ('W', 13), ('W', 6), ('W', 2
), ('W', 0)]
INFO:     127.0.0.1:60982 - "POST /read HTTP/1.1" 200 OK
```

Figure 6: Pattern of accesses into the untrusted database when reading the article titled "A". 'R' and 'W' represent reads and writes, respectively. The numbers represent the keys in the database that are accessed. Notice the tree structure that Path ORAM imposes on the database. Each call to `Access` causes a series of reads along the path from the root to a leaf (which we can see by the fact that the initial read is at 0, which is the root, and each read to an index i is followed either $2i+1$ or $2i+2$, which are the children). This is followed by a series of writes backward along the same path. Since the article "A" is stored across two blocks, this happens twice. The privacy guarantees come from the fact that the leaf nodes that are accessed are uniformly random. Running the same query again will result in a different sequence of accesses.

challenging system-level integration issues, we were unable to combine them effectively into a unified solution. Specifically, the integration of the RAG system within a TEE environment encountered dependency-related obstacles.

7 Contributions and Conclusion

As this is a final project, we wish to directly address the expectation to present results and evaluations clearly. Although we did not integrate the entire private RAG into a single system, to enable results such as profiling for speed or memory usage, we wish to now highlight several of the results we produced.

We now summarize the primary contributions of this final project.

- In Section 4, we provide formal definitions for Private RAG systems.
- In Section 6.2 and in the Github repo (Wiki-RAG Github), we provide clear and reproducible instructions for deploying a server within an AWS Nitro Trusted Execution Environment (TEE).
- In Section 6.1 and in the aforementioned Github repo, we developed and open-sourced the first Wikipedia-based RAG system .
- In section 6.3 we identify and document critical issues within existing PathORAM implementations.
- In section 6.3 and in a second Github repository we implement PathORAM formally in Dafny with correctness verification.
- Lastly, in Github, we produce a functional Wikipedia PathORAM prototype in Python. Both the Dafny and Python implementations are available at Private-RAG Github.

7.1 Future Work

The immediate future work is focused on integrating the three individually developed components, the RAG system, the TEE deployment framework, and the PathORAM protocol, into a single, cohesive and fully operational secure system. Currently, the reason why we were unable to do this in practice is not due to theoretical constraints but rather solely due to dependency issues and build issues. The primary issue that we would like to resolve is to resolve these issues and integrate the system into 1, running on an AWS EC2 instance. After that, we would do profiling work to measure potential speed ups.

We note that when consulting experts in the field (PhDs and professors we spoke to), a common critique we heard is along the lines of "Trusted Execution Environments have many side-channel attacks". We would be interested in exploring what concrete side-channel attacks TEEs are particularly susceptible to, and if there are ways to produce a patchwork of protection to prevent these.

References

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query cost. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 962–979. IEEE, 2018.

[ACS18] Sebastian Angel, Hao Chen, and Srinath Setty. Making pir practical for databases. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2018.

[BMP11] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious ram practical. 2011.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[CLP14] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure oram with overhead. In *International Conference on the*

Theory and Application of Cryptology and Information Security, pages 62–81. Springer, 2014.

[Con22] Graeme Connell. Technology deep dive: Building a faster oram layer for enclaves. *Signal Blog*, August 2022.

[CP13] Kai-Min Chung and Rafael Pass. A simple oram. *Cryptology ePrint Archive*, 2013.

[CZW$^+$24] Yihang Cheng, Lan Zhang, Junyang Wang, Mu Yuan, and Yunhao Yao. Remoterag: A privacy-preserving llm cloud rag service. *arXiv preprint arXiv:2412.12775*, 2024.

[DGD$^+$25] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library, 2025.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious ram without random oracles. In *Theory of Cryptography: 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings 8*, pages 144–163. Springer, 2011.

[GM11] Michael T Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *International Colloquium on Automata, Languages, and Programming*, pages 576–587. Springer, 2011.

[GMOT11] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 95–100, 2011.

[GMOT12] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Practical oblivious storage. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 13–24, 2012.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. In *Journal of the ACM (JACM)*, volume 43, pages 431–473. ACM New York, NY, USA, 1996.

[Lei10] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic Based Program Synthesis and Transformation: 19th International Symposium, LOPSTR 2009, Coimbra, Portugal, September 9-11, 2009, Revised Selected Papers 19*, pages 348–360. Springer, 2010.

[Lin17] Yehuda Lindell. How to simulate it–a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pages 277–346, 2017.

[LPM$^+$13] Jacob R Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to {Large-Scale} data in the data center. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 199–213, 2013.

[LPP$^+$20] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474, 2020.

[LZY$^+$23] Xiaoguo Li, Bowen Zhao, Guomin Yang, Tao Xiang, Jian Weng, and Robert H Deng. A survey of secure computation using trusted execution environments. *arXiv preprint arXiv:2302.12150*, 2023.

[Nuc25] Nuclia. Rag as a service - retrieval augmented generation, 2025. Accessed: 23-March-2025.

[ORR21] Sam Olsen, Peter Rindal, and Mike Rosulek. Spiral: Fast, high-rate single-server pir via fhe composition. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*, page 2931–2947, New York, NY, USA, 2021. Association for Computing Machinery.

[Rag25] Ragie. Ragie.ai: Rag api for search and content enhancement, 2025. Accessed: 23-March-2025.

[SDS$^+$18] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. *Journal of the ACM (JACM)*, 65(4):1–26, 2018.

[SMS$^+$22] Moritz Schneider, Ramya Jayaram Masti, Shweta Shinde, Srdjan Capkun, and Ronald Perez. Sok: Hardware-supported trusted execution environments. *arXiv preprint arXiv:2205.12742*, 2022.

[SSS11] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. *arXiv preprint arXiv:1106.3652*, 2011.

[ZFY25] Pengcheng Zhou, Yinglun Feng, and Zhongliang Yang. Privacy-aware rag: Secure and isolated knowledge retrieval. *arXiv preprint arXiv:2503.15548*, 2025.

[ZPZP24] Jinhao Zhu, Liana Patel, Matei Zaharia, and Raluca Ada Popa. Compass: Encrypted semantic search with high accuracy. *Cryptology ePrint Archive*, 2024.

[ZPZS24] Mingxun Zhou, Andrew Park, Wenting Zheng, and Elaine Shi. Piano: extremely simple, single-server pir with sublinear server computation. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4296–4314. IEEE, 2024.

A Implementing a RAG on Wikipedia

`Wiki-RAG` provides a streamlined method for downloading the entirety of Wikipedia and incorporating it into a Retrieval-Augmented Generation (RAG) system. The primary benefit is an entirely offline setup, avoiding repeated external API calls. After retrieving the title of a relevant article through the RAG, the complete article content can be fetched from either a local offline store or directly from Wikipedia if desired.

The specific Wikipedia data used was downloaded on April 10, 2025, from Wikimedia.

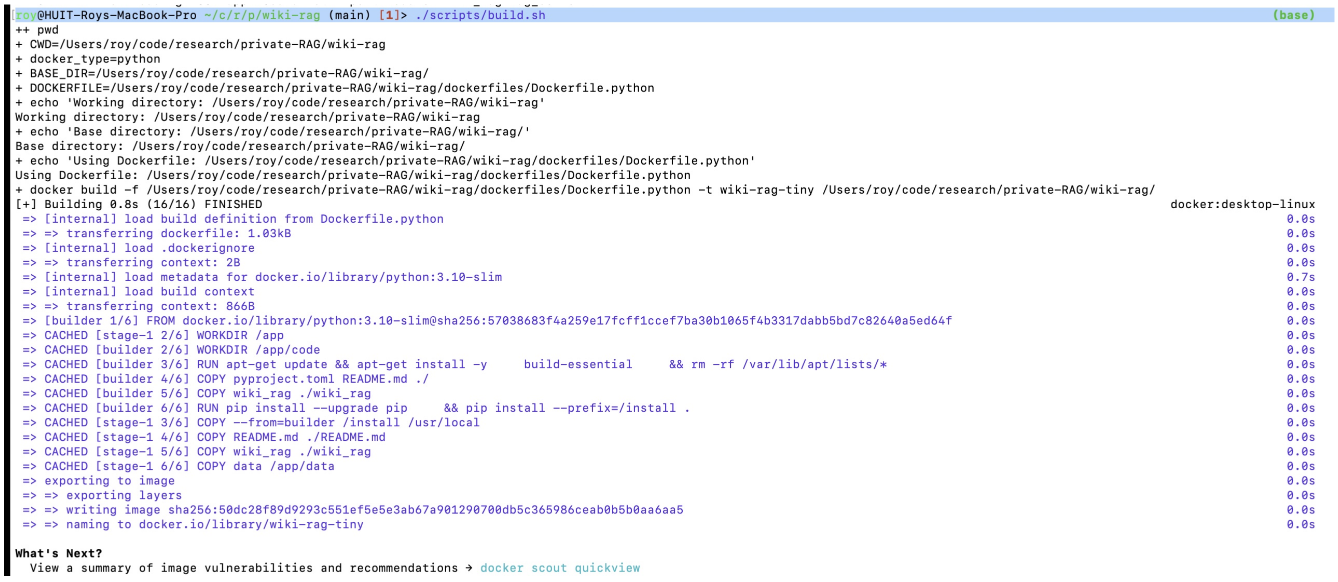
The pre-built RAG is publicly available on HuggingFace.

A.1 Embedding Approach

The provided RAG employs embeddings generated from each complete Wikipedia article. Initial tests using smaller article segments suggested that embeddings created from full articles yielded superior retrieval accuracy.

A.2 Quick Start Guide

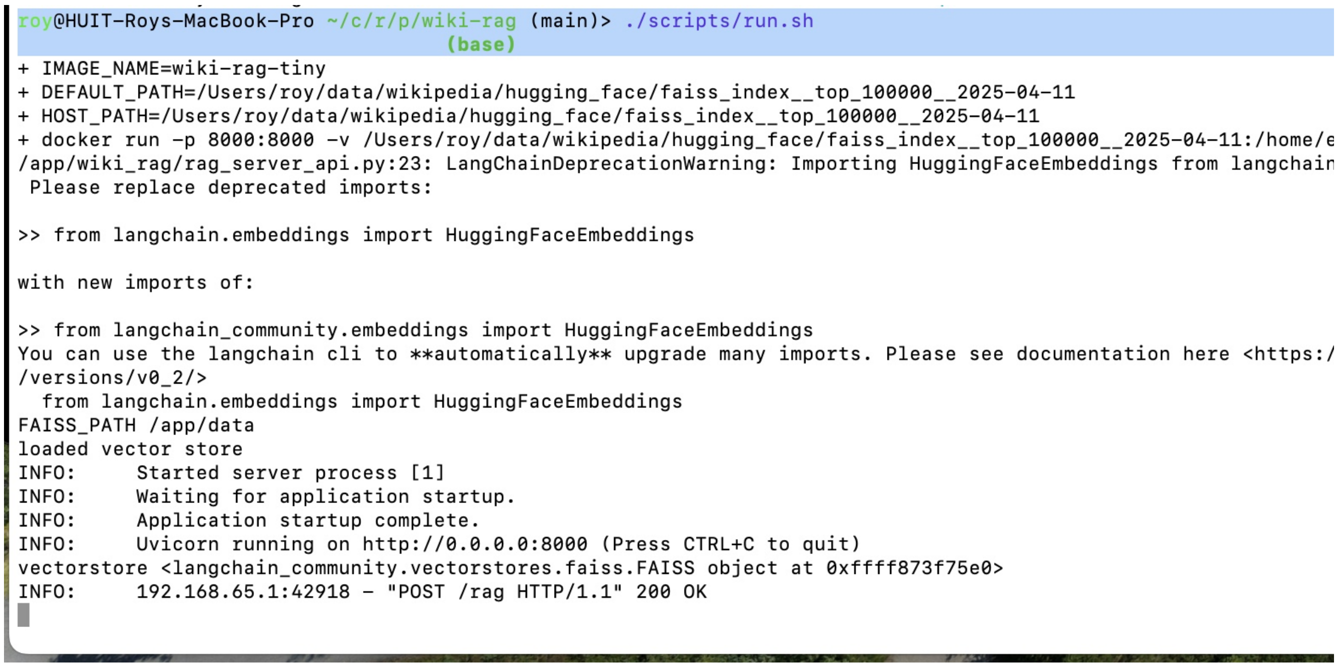
We provide a guide for using this described in the github repo. Here is documentation of the RAG system working locally:



```
roy@HUIT-Roys-MacBook-Pro ~/c/r/p/wiki-rag (main) [1]> ./scripts/build.sh                                                                    (base)
++ pwd
+ CWD=/Users/roy/code/research/private-RAG/wiki-rag
+ docker_type=python
+ BASE_DIR=/Users/roy/code/research/private-RAG/wiki-rag/
+ DOCKERFILE=/Users/roy/code/research/private-RAG/wiki-rag/dockerfiles/Dockerfile.python
+ echo 'Working directory: /Users/roy/code/research/private-RAG/wiki-rag'
Working directory: /Users/roy/code/research/private-RAG/wiki-rag
+ echo 'Base directory: /Users/roy/code/research/private-RAG/wiki-rag/'
Base directory: /Users/roy/code/research/private-RAG/wiki-rag/
+ echo 'Using Dockerfile: /Users/roy/code/research/private-RAG/wiki-rag/dockerfiles/Dockerfile.python'
Using Dockerfile: /Users/roy/code/research/private-RAG/wiki-rag/dockerfiles/Dockerfile.python
+ docker build -f /Users/roy/code/research/private-RAG/wiki-rag/dockerfiles/Dockerfile.python -t wiki-rag-tiny /Users/roy/code/research/private-RAG/wiki-rag/
[+] Building 0.8s (16/16) FINISHED                                                                                              docker:desktop-linux
 => [internal] load build definition from Dockerfile.python                                                                                     0.0s
 => => transferring dockerfile: 1.03kB                                                                                                          0.0s
 => [internal] load .dockerignore                                                                                                               0.0s
 => => transferring context: 2B                                                                                                                 0.0s
 => [internal] load metadata for docker.io/library/python:3.10-slim                                                                             0.7s
 => [internal] load build context                                                                                                               0.0s
 => => transferring context: 866B                                                                                                               0.0s
 => [builder 1/6] FROM docker.io/library/python:3.10-slim@sha256:57038683f4a259e17fcff1ccef7ba30b1065f4b3317dabb5bd7c82640a5ed64f                0.0s
 => CACHED [stage-1 2/6] WORKDIR /app                                                                                                           0.0s
 => CACHED [builder 2/6] WORKDIR /app/code                                                                                                      0.0s
 => CACHED [builder 3/6] RUN apt-get update && apt-get install -y     build-essential     && rm -rf /var/lib/apt/lists/*                        0.0s
 => CACHED [builder 4/6] COPY pyproject.toml README.md ./                                                                                       0.0s
 => CACHED [builder 5/6] COPY wiki_rag ./wiki_rag                                                                                               0.0s
 => CACHED [builder 6/6] RUN pip install --upgrade pip     && pip install --prefix=/install .                                                   0.0s
 => CACHED [stage-1 3/6] COPY --from=builder /install /usr/local                                                                                0.0s
 => CACHED [stage-1 4/6] COPY README.md ./README.md                                                                                             0.0s
 => CACHED [stage-1 5/6] COPY wiki_rag ./wiki_rag                                                                                               0.0s
 => CACHED [stage-1 6/6] COPY data /app/data                                                                                                    0.0s
 => exporting to image                                                                                                                          0.0s
 => => exporting layers                                                                                                                         0.0s
 => => writing image sha256:50dc28f89d9293c551ef5e5e3ab67a901290700db5c365986ceab0b5b0aa6aa5                                                    0.0s
 => => naming to docker.io/library/wiki-rag-tiny                                                                                                0.0s

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout quickview
```

Figure 7: Example of how to build RAG in docker image



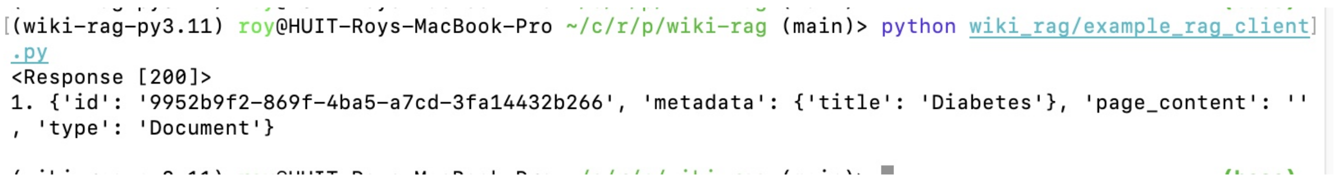
```
roy@HUIT-Roys-MacBook-Pro ~/c/r/p/wiki-rag (main)> ./scripts/run.sh
                                    (base)
+ IMAGE_NAME=wiki-rag-tiny
+ DEFAULT_PATH=/Users/roy/data/wikipedia/hugging_face/faiss_index__top_100000__2025-04-11
+ HOST_PATH=/Users/roy/data/wikipedia/hugging_face/faiss_index__top_100000__2025-04-11
+ docker run -p 8000:8000 -v /Users/roy/data/wikipedia/hugging_face/faiss_index__top_100000__2025-04-11:/home/e
/app/wiki_rag/rag_server_api.py:23: LangChainDeprecationWarning: Importing HuggingFaceEmbeddings from langchain
 Please replace deprecated imports:

>> from langchain.embeddings import HuggingFaceEmbeddings

with new imports of:

>> from langchain_community.embeddings import HuggingFaceEmbeddings
You can use the langchain cli to **automatically** upgrade many imports. Please see documentation here <https:/
/versions/v0_2/>
  from langchain.embeddings import HuggingFaceEmbeddings
FAISS_PATH /app/data
loaded vector store
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
vectorstore <langchain_community.vectorstores.faiss.FAISS object at 0xffff873f75e0>
INFO:     192.168.65.1:42918 - "POST /rag HTTP/1.1" 200 OK
```

Figure 8: Example of how to run RAG inside of Docker image



```
(wiki-rag-py3.11) roy@HUIT-Roys-MacBook-Pro ~/c/r/p/wiki-rag (main)> python wiki_rag/example_rag_client
.py
<Response [200]>
1. {'id': '9952b9f2-869f-4ba5-a7cd-3fa14432b266', 'metadata': {'title': 'Diabetes'}, 'page_content': ''
, 'type': 'Document'}
```

Figure 9: Example of simple client making local API call to RAG running inside docker image.

A.3 Detailed Repository Setup

The process for creating the `Wiki-RAG` repository includes the following steps:

1. **Downloading Wikipedia:** Approximately 22 GB in size, requiring around 2 hours using Wget or 30 minutes using Aria2c:

    ```
    aria2c -x 16 -s 16 https://dumps.wikimedia.org/enwiki/latest/
    enwiki-latest-pages-articles.xml.bz2
    ```

2. **Extracting Wikipedia into JSON format:**

    ```
    python3 WikiExtractor.py enwiki-latest-pages-articles.xml.bz2 \
    -o extracted --json
    ```

    (A provided SLURM script, `extract_wiki.slrm`*, automates this step with some hard-coded parameters.)*

3. **Selecting top Wikipedia articles:** Obtain the list of top 100K or 1M articles by page views from:

    ```
    https://dumps.wikimedia.org/other/pageviews/2024/2024-12/
    ```

4. **Constructing the FAISS index:** Use provided helper scripts in `wiki_rag` directory:

```
    wiki_rag/construct_faiss.py
```

(The associated SLURM script, `construct_faiss.slrm`*, automates this.)*

5. **Building Docker Image:** The Docker image embeds the FAISS index to serve a simple retrieval API. *(Note: The FAISS index must be pre-generated and placed in the* `data` *directory prior to building.)*

A.4 Repository Structure

The core file structure of the `Wiki-RAG` repository is as follows:

```
wiki_rag
 __init__.py
 construct_faiss.py      # Builds FAISS index from Wikipedia
 rag.py                  # Helper functions for FAISS
 rag_server.py           # Serves Wikipedia entries via FAISS
 example_rag_client.py   # Simple client to query the RAG server
 wikipedia.py            # Utilities for interacting with Wikipedia data
```

A.5 Quick Download of Prebuilt Wiki-RAG FAISS Index

To quickly download the prebuilt RAG index:

```
#!/bin/bash

REPO="royrin/wiki-rag"
FOLDER="faiss_index__top_100000__2025-04-11__title_only"

FILES=$(curl -s https://huggingface.co/api/models/$REPO \
        | jq -r '.siblings[].rfilename')
FILES_TO_DOWNLOAD=$(echo "$FILES" | grep "^$FOLDER/")

mkdir -p $FOLDER
cd $FOLDER

for FILE in $FILES_TO_DOWNLOAD; do
    echo "Downloading $FILE"
    mkdir -p "$(dirname \"$FILE\")"
    curl -L -o "$FILE" \
    "https://huggingface.co/$REPO/resolve/main/$FILE"
done
```

B Setting Up RAG in AWS Nitro TEE

In our README in Wiki-RAG Github, we provide in-depth details for how to set up the AWS Nitro TEE.