CSHER: A System for Compact Storage with HE-Retrieval

Adi Akavia
University of Haifa

Neta Oren
University of Haifa

Boaz Sapir
Intuit Israel Inc.

Margarita Vald
Intuit Israel Inc.

Abstract

Homomorphic encryption (HE) is a promising technology for protecting data in use, with considerable progress in recent years towards attaining practical runtime performance. However, the high storage overhead associated with HE remains an obstacle to its large-scale adoption. In this work we propose a new storage solution in the two-server model *resolving the high storage overhead associated with HE*, while preserving rigorous data confidentiality. We empirically evaluated our solution in a proof-of-concept system running on AWS EC2 instances with AWS S3 storage, demonstrating storage size with *zero overhead* over storing AES ciphertexts, and 10μs amortized end-to-end runtime. In addition, we performed experiments on multiple clouds, i.e., where each server resides on a different cloud, exhibiting similar results. As a central tool we introduce the *first perfect secret sharing scheme with fast homomorphic reconstruction over the reals*; this may be of independent interest.

1 Introduction

The privacy and safety of individuals and organizations are threatened by the ubiquity of data collected by products and services as part of the so-called AI revolution. Data collecting companies are driven to address this threat to avoid reputational damage in case of a data breach and to comply with privacy regulations such as GDPR [4] and CCPA [18]. A promising approach for enhancing data protection is to use secure computation (MPC) [36, 60] and homomorphic encryption (HE) [31, 55] that support data protection at all times, not only at rest and in transit, but also for data in use. The HE approach is particularly appealing due to its compatibility with existing dataflow. It supports computations over encrypted data by stand-alone servers (computing server) requiring no interaction, and the complexity of all other parties, if any exists, is independent of the complexity of the evaluated function (unlike in MPC). Indeed there has been much progress towards attaining practical runtime for HE-based solutions, e.g., for privacy preserving machine learning (PPML) [6, 7, 13, 14, 24, 34, 35, 38, 39, 42, 43, 48, 54, 56]. These solutions often operate over massive datasets (e.g., in training machine learning models) that must be stored in HE-encrypted form. More generally, as HE-based solutions are becoming faster and faster (due to improvements in schemes, algorithms and hardware), more data is likely to be HE-encrypted and stored.

The high storage overhead associated with HE encryption however is a significant obstacle preventing large-scale adoption. Concretely, storing HE ciphertexts of state-of-the-art HE schemes and implementations (e.g., the CKKS scheme [23] in Microsoft's SEAL library [57]) incurs a $10\times$ to $10^4\times$ blowup in storage size compared to storing the data in cleartext or encrypted using standard schemes such as AES. The blowup rate depends on the number of data items packed in each ciphertext. The $10\times$ blowup occurs at maximal packing capacity, which is not always applicable. For example, packing cannot be fully utilized if data must be encrypted in a streaming fashion, or arrives from distinct sources with few data items each, or the computation is not suitable for entry-wise vector operations as supported on packed data. Furthermore, although storage costs are rapidly declining, for some of the use cases being proposed for HE, such as secure computation within an enterprise, the relevant storage is entire enterprise-level datalakes at petabyte scale or more. At such scale, even the low-end of $10\times$ blowup in storage cost is clearly unacceptable from both a financial and operational perspective.

The question initiating this research is therefore:
Can the high storage overhead of HE be eliminated?

1.1 Our Contribution

This work affirmatively resolves the above question by proposing: (1) A new storage solution in the two-server model eliminating the high storage overhead associated with HE. (2) A proof-of-concept system, named CSHER, demonstrating the appealing storage-size and runtime performance attained by our solution. (3) As a central tool we introduce the first perfect secret sharing scheme supporting fast homomorphic

reconstruction over the reals.

Contribution 1: Compact storage with HE-retrieval. We present a new storage and retrieval solution that supports retrieving data in HE-encrypted form without revealing any information about the underlying data (HE-retrieval), while producing storage of size as small as when storing data encrypted by standard encryption schemes such as AES (compact storage) and maintaining fast end-to-end storage and retrieval runtime, only twice the time of directly storing and retrieving HE ciphertexts. Our solution is generic and can be instantiated with any public key HE scheme supporting additive homomorphism, where the plaintext addition may be over a finite ring as in Paillier [53] and BGV [17] or over the reals as in CKKS [23].

Our solution consists of two protocols, *store* and *retrieve*. *Store* is a non-interactive protocol allowing multiple data producers to upload data to a common storage platform (e.g., a datalake on AWS S3), where data can be uploaded in blocks or one-by-one, possibly over time and interleaved with retrieval queries. *Retrieve* is a single-round interactive protocol between the computing server that obtains as output the HE encrypted data and an auxiliary service that has no output. Security holds in the two-server model, i.e., when the computing server and auxiliary service are non-colluding. In this model we guarantee correctness against semi-honest adversaries and privacy against malicious ones.

Our solution can be combined with any homomorphic computation on the HE-retrieved data, simply by plugging in the retrieved HE ciphertexts as input to the homomorphic computation ("retrieve-then-evaluate"). Importantly, the homomorphic computation is executed solely by the computing server, without any help from the auxiliary service. The overall communication complexity in the integrated retrieve-then-evaluate protocol is independent of the complexity of the evaluated function.

To support integration with a wide range of computations, our solution supports *dynamic control* at the time of data retrieval for: (i) *Data:* fine-grained dynamic choice of the data items to be retrieved is supported, allowing retrieval of any subset of the data items. (ii) *HE scheme and parameters:* dynamic choice of the HE scheme with which the data is encrypted, its public key and context parameters such as the degree of the cyclotomic polynomial, is likewise supported. (iii) *Batching:* flexibly choosing which data items are packed together in each ciphertext is supported. The flexible dynamic control allows tailoring each HE-retrieval to best optimize the computation at hand; for example, dynamically choosing a high degree cyclotomic polynomial in CKKS to maintain correctness if integrating with a deep homomorphic computation, and low degree otherwise to speedup performance.

This offers a viable industry compatible solution for computing on HE-encrypted data with zero data exposure during the pipeline while eliminating the high storage overhead associated with storing HE ciphertexts.

Discussion: why use HE if we have two-servers? A major advantage of HE-based secure computation, in comparison to secret sharing or garbling based solutions, is its communication complexity: with HE, communication is typically proportional only to the size of the input and output ciphertexts; in contrast, with secret sharing and garbling, it is proportional to the size of the evaluated circuit (i.e., the complexity of the underlying computation) on top of its input and output size. Our retrieve-then-evaluate solution achieves the same advantage: it has a communication complexity that is proportional only to the input and output size while being independent of the size of the evaluated circuit. We note further that the auxiliary service has a predefined lightweight logic. This is in sync with existing enterprise best practices, where services with a simple predefined logic (e.g., a key management or our auxiliary service) can be effectively safeguarded, making the non-collusion assumption credible.

Contribution 2: Proof-of-concept system. We present CSHER – a proof-of-concept system implementing our compact storage with HE-retrieval while using CKKS [23] as the HE scheme and our secret sharing scheme as a central tool.

We implemented the system using Amazon Web Services (AWS) with separate EC2 instances (M5.2xlarge) for each participating entity (data producer, computing server, and auxiliary service). The instances communicate via HTTP. We use an S3 bucket for storage. We enforced access control to S3 by applying AES encryption on uploaded data, with the decryption key accessible to authorized parties. In addition, we implemented our system in a multi-cloud environment which is identical to the one on AWS except that the computing server is running on a Google Cloud n2-standard-8 instance (rather than on AWS). Separating the servers into two competing cloud providers is aimed at increasing the confidence that they do not collude.

Ous system is compatible with the industry's most commonly used architecture, tools and best-practices for storing and processing sensitive big data. For example, we support the use of standardized encryption schemes for data protection in long-term storage (e.g., AES), native datatypes in datalake storage (e.g., double), and commonly used tools for computing servers and datalake (e.g., AWS EC2 instances with S3 storage). More broadly, incorporating our system into exiting pipelines requires minimal local expansion of current systems, supporting transparent integration with existing applications and data processing flows, e.g., by dynamically choosing whether data is retrieved in cleartext or in HE form. Importantly, our system does not only comply with existing architectures but simultaneously improves on it: to compute on data, current flows read it from datalake in cleartext form, whereas our system supports transformation from AES ciphertexts to CKKS (or other schemes of the user's choice) to be fed into the homomorphic computation with zero data exposure throughout the entire pipeline.

We ran extensive experiments with empirical results demon-

strating that our solution achieves:

1. **Compact storage** with zero overhead over using AES encryption to protect sensitive data; and $10\times$ to $10^4\times$ improvement in storage size over directly storing CKKS encrypted data.

2. **Fast runtime** of end-to-end storage plus HE-retrieval, only twice the time of directly storing plus retrieving CKKS encrypted data. For example, the overall runtime for storing and then retrieving 2,031,616 data items is 19.8s in our solution compared to 10.3s in the baseline solution of directly storing and retrieving HE ciphertexts.

Furthermore, we demonstrate our system's support for flexible choice of the CKKS parameters by dynamically setting the degree of the cyclotomic polynomial. Likewise, the data to be retrieved and how it is packed (batched) into CKKS ciphertexts can be flexibly set at the time of retrieval.

Concretely, on *two-million* data items, the storage size in our solution is 16 MB (cf. 168 MB in the baseline experiment of directly storing and retrieving batched CKKS ciphertexts with Z-standard library for compression), the storage runtime is 2s (cf. 7.8s), the HE-retrieval runtime is 17.8s (cf. 2.5s), and communication size and time is 331 MB and 0.25s. The amortized performance per data item is therefore: 8 bytes of storage, 1μs and 9μs for storage and retrieval runtime respectively, and 165 bytes and 0.1μs communication. The performance of our multi-cloud system are essentially identical, except for a small overhead in communication time when the communication is between AWS and Google Cloud rather than between two AWS instances; concretely, the overhead is 0.23s (0.14μs when amortized per data item). We note that the HE-retrieval runtime is minor in comparison to the homomorphic computation to be applied on the retrieved encrypted data. E.g., even when integrated with a relatively modest computation –homomorphic evaluation of a depth four decision tree on the retrieved ciphertexts– only 7% of the computation runtime was devoted to HE-retrieval (see Table 3). Moreover, the communication complexity is the size of the retrieved ciphertexts, while being independent of the complexity of the homomorphic evaluation.

These results empirically demonstrate the practical usability of our system. We note that our implementation was single threaded and sequential, which does not utilize the fact that our solution is embarrassingly parallelizable; extending our implementation to incorporate parallelization and streaming should gain a major further speedup.

Contribution 3: Secret sharing with fast homomorphic reconstruction over the reals. We present the first perfect secret sharing scheme supporting fast homomorphic reconstruction over the reals. The scheme offers a 2-out-of-2 secret sharing for secrets in $[0,1]$.[1] Homomorphic reconstruction requires only additive homomorphism over the reals (as supported for example by CKKS), when given one encrypted share and the other in cleartext. In contrast, in all prior secret sharing schemes, the reconstruction algorithm is either computed over a finite field or ring (e.g., [9, 11, 15, 20–22, 27, 29, 45, 58]); or involves operations such as reducing numbers modulo one [12, 26] that are impractical for homomorphic computations even with state-of-the-art techniques [40]; or does not achieve perfect security [28, 59].

Theorem 1.1 (secret sharing, informal)**.** *There exists a 2-out-of-2 perfect secret sharing scheme for* $[0,1]$ *whose reconstruction algorithm is a degree* 1 *polynomial over the reals in the 2nd share as the unknown. Consequently, it can be homomorphically evaluated, over an encrypted 2nd share, by any scheme supporting additive homomorphism over the reals.*

1.2 Overview of our Construction

We overview the main ideas and tools in our construction.

Store secret shares, homomorphically reconstruct to retrieve HE-ciphertexts. Our starting point is the following idea. To store data, first secret share it using a 2-out-of-2 secret sharing scheme, and store the 1st (resp. 2nd) share with access authorized to the computing (resp. auxiliary) server. To retrieve the data in HE-encrypted form, the auxiliary server encrypts its share using the HE scheme and sends the ciphertext to the computing server; the computing server homomorphically reconstructs the data, using his cleartext share together with the received encrypted share, to obtain the data in HE-encrypted form. To instantiate the above idea we require a "HE-friendly secret sharing" whose reconstruction algorithm can be efficiently computed over encrypted input. For HE schemes over finite rings such as Paillier [53] and BGV [17], additive secret sharing is friendly, as its reconstruction requires only additive homomorphism over the finite ring. In contrast, for HE over the reals, no friendly secret sharing scheme was known prior to our work.

Friendly secret sharing for arithmetic over the reals. In our secret sharing scheme, the key idea for achieving fast homomorphic reconstruction over the reals is to avoid the mod 1 step of [12, 26]. In [12, 26], sharing a number $x \in [0,1)$ is by sampling a random $t \in [0,1)$ and outputting shares $s_1 = t$ and $s_2 = x + t \bmod 1$; reconstructing is by outputting $s_1 + s_2 \bmod 1$. In our secret sharing for reals we avoid computing mod 1 during reconstruction. Instead, to share a secret $x \in [0,1]$ we sample uniformly random $(b,t) \in \{0,1\} \times [0,1)$, and compute shares: $b+t$ and $x+(b+t) \bmod 2$. We represented each share by the pair of its integral of fractional part: $s_1 = (b,t)$ and $s_2 = (s_{int}, s_{frac})$ for $s_{int} = \lfloor x+t \rfloor + b \bmod 2$ and $s_{frac} = ((x+b+t) \bmod 2) - s_{int}$. Equivalently, $s_{int} = (-1)^b(\lfloor x+t \rfloor - b)$ and $s_{frac} = (x+t) - \lfloor x+t \rfloor$. Our reconstruction outputs $s_{frac} - t + (-1)^b s_{int} + b$ (all operations are computed over the reals). The key property of our scheme

[1] Other ranges can be addressed via scaling and shifting to $[0,1]$.

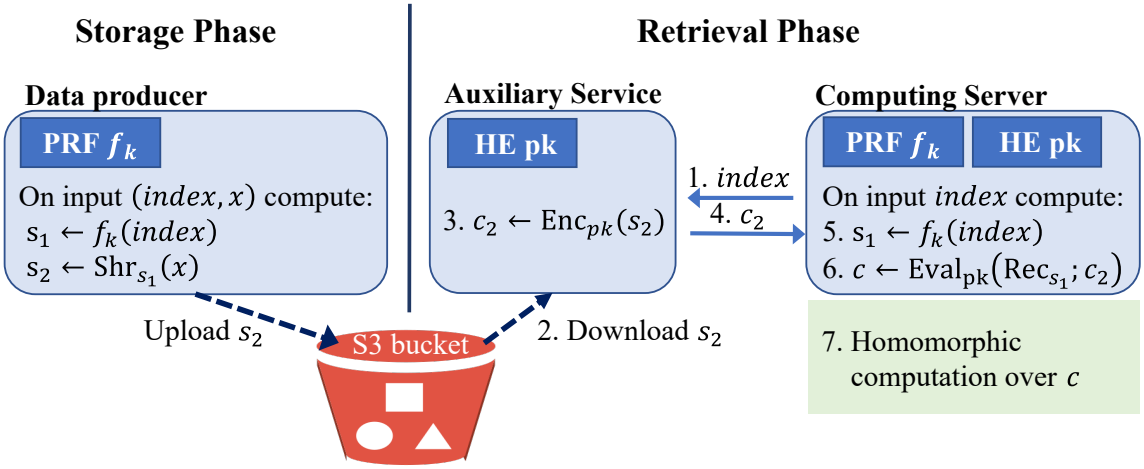


Figure 1: Our protocol (Figure 4): storage (left), HE-retrieval (right), subsequent homomorphic computation (green). The protocol employs a PRF f_k to generate $s_1 = (b,t)$; a secret sharing scheme with 1st share hardwired to be s_1, denoted $(\mathsf{Shr}_{s_1}, \mathsf{Rec}_{s_1})$; and an HE scheme with which we encrypt (entry-by-entry) $s_2 = (s_{int}, s_{frac})$ and homomorphically evaluate Rec_{s_1} on the encrypted s_2.

is that, when given cleartext b,t and encrypted s_{int}, s_{frac}, as in our HE-retrieval protocol, the reconstruction algorithm is a degree 1 polynomial in s_{int}, s_{frac}, and therefore it can be efficiently evaluated using any additive homomorphic encryption over the reals, e.g., CKKS. The security of our secret sharing scheme essentially follows by noticing that s_{frac} is a masking of x by adding to it a random $t \in [0,1)$ and reducing modulo 1, and s_{int} is a masking of the integral part of $x+t$ by XOR-ing it with a random bit b. The key point however is that we never have to compute this reduction modulo 1 during reconstruction.

Storage optimized to zero overhead. To attain zero overhead over storing the data in cleartext, instead of storing both shares we store only the 2nd share (s_2), while generating the 1st share (s_1) on-the-fly, pseudo-randomly, using a pseudorandom function f_k whose key k is accessible to the data producer and computing server, but not the auxiliary server. We use a single key k across all store and retrieve sessions. This works because the 1st share in our secret sharing for reals is chosen uniformly at random and independently of the secret (likewise, in additive secret sharing over a finite ring). Since the share size in our secret sharing for reals (likewise, in additive secret sharing over a finite ring) has the same size as the cleartext data, the storage size is identical to storing the data in cleartext. See an illustration in Figure 1.

1.3 Related Work

Storage systems utilizing secret sharing. Secret sharing is used in the context of storage systems to ensure data availability and confidentiality, see a survey in [46] Section 4.6.9. But it has not been used for HE-retrieval prior to our work. The shares' size is a major consideration in the context of storage systems due to its implications to storage costs. Secret sharing based storage solutions typically employ Krawczyk's computational secret sharing scheme [44] that has nearly optimal share size: the total size of the shares $\sum_{i=1}^{n} |s_i| = |x| + n|K|$ for $|x|$ the data size, $|K|$ the size of a symmetric encryption key freshly generated for each data item, and n the number of shares. All solutions use secret sharing over a finite field or ring (see a survey in [10]). In contrast, in our work we propose a new secret sharing scheme for storage that is over real numbers. The total size of the shares in our scheme is $|x| + |K|$, consisting of the data x and a key K (cf. $|x| + 2|K|$ in [44]); moreover, the key is never revealed to the other party in our compact storage solution, and therefore we can use a single persistent key for all data (rather than requiring a fresh key to be generated and stored for each data item in [44]).

Rate-1 HE. Gentry and Halevi [32] and Brakerski et al. [16] designed a highly effective technique for packing many data items in an HE ciphertext achieving zero overhead in ciphertexts size over plaintext size. The problem however is that these highly packed cipehrtexts support only additive homomorphism, which considerably limits their usability.

Homomorphic decryption for symmetric ciphers. Naehrig et al. [52] suggested transforming AES ciphertexts into HE ciphertexts via homomorphic evaluation of AES decryption when given as input a HE encryption of the AES decryption key. This was followed by a line of work for AES ciphers [30, 33] and other symmetric ciphers such as LowMC [8], Kreyvium [19], FLIP and FiLIP [50, 51], RASTA and MASTA [37, 51], and HERA [25] (see a survey [5]). However, this approach currently offers a running time that is not practical for handling massive amounts of data. Moreover, using non standardized ciphers does not comply with industry practices of using AES. Our solution is at least $10^4\times$ faster

than the above works, albeit in the two-server model.

Homomorphic repacking. An alternative approach to proving a dynamic choice of the batching pattern in the retrieved HE ciphertext is to use homomorphic packing or re-packing. I.e., given HE ciphertexts, homomorphically decrypt them and re-organize their underlying messages as to produce a ciphertext containing the desired messages in the desired batching pattern. The runtime of homomorphic packing is at least the time for homomorphic decryption (aka, bootstrapping), which takes over 20s for CKKS ciphertexts with the current state-of-art (see [47, Table 3]), and the runtime for repacking 4096 items is 60.69s by [49, Table VIII]. In contrast, our solution takes 0.06s for 4096 data items (see Table 2).[2]

Follow-up work. Zhou et al. [61] implemented our generic protocol on data types and HE schemes beyond those in our empirical evaluation, demonstrating its good performance and high usability. They deviate from our solution in two main points. First, they use a binary representation of real numbers, where the complexity of their reconstruction algorithm (i.e., the number of transmitted ciphertexts and homomorphic operations) is linear in this binary representation length (cf. $O(1)$ in our solution). Second, their system modeling consists of a client and server that must interact during both the storage and the retrieval phases (cf. our system modeling that is consistent with industry systems where data is uploaded to storage by data producers and fetched by data consumers).

Paper Organization

The rest of this paper is organized as follows: preliminary terminology and definitions appear in Section 2; our secret sharing scheme in Section 3; our compact storage for HE-retrieval in Section 4; the implemented system and empirical evaluation in Section 5; and conclusions in Section 6.

2 Preliminaries

We use standard definitions for functions being *negligible* with respect to a system parameter λ called the *security parameter*, denoted $\mathsf{neg}(\lambda)$; similarly for *polynomial*, where ppt stands for *probabilistic polynomial time* in λ. We follow standard definitions for *probability ensembles* and *computationally indistinguishability*, denoted $\approx_c$; when two distributions are identical we denote this by $\equiv$. We denote by $s \leftarrow_R S$ a uniformly random sample s from a set S. See the formal definitions in [41]. We use $|z|$ to denote the binary representation length of z. We consider both fixed-point and floating point binary representations of real numbers in $[0,1]$. We denote by $Q1.p$ the fixed-point representation using 1 bit for the integral part and p bits for the fractional part. For a floating point we define the *precision of the significand* (respectively, *exponent*) to be the number of bits allocated by the format for the significant (respectively, exponent). That is, for a floating point in *double* precision, the precision of the significand is 53 bits (11 bits for the exponent).

Definition 2.1 (secret sharing)**.** *A* 2-out-of-2 secret sharing *scheme for* A *is a pair of* ppt *algorithms* $(\mathsf{Shr}, \mathsf{Rec})$ *such that:*

- Shr *is a randomized algorithm that given* $x \in \mathsf{A}$ *outputs a pair of shares* (s_1, s_2).
- Rec *is a deterministic algorithm that given a pair of shares* (s_1, s_2) *outputs an element in* A.

The correctness *requirement is that for all* $x \in \mathsf{A}$,

$$\mathsf{Rec}(\mathsf{Shr}(x)) = x.$$

The (perfect) security *requirement is that for every* $x, x' \in \mathsf{A}$ *and* $i \in \{1,2\}$, *the following two distributions are identical:*

$$\{s_i\}_{(s_1,s_2)\leftarrow \mathsf{Shr}(x)} \equiv \{s'_i\}_{(s'_1,s'_2)\leftarrow \mathsf{Shr}(x')}.$$

Definition 2.2 (homomorphic encryption (HE))**.** *A homomorphic public-key encryption (HE) scheme* $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ *with message space* $\mathcal{M}$ *is a quadruple of* ppt *algorithms as follows:*

- Gen *(key generation) takes as input the security parameter* 1^λ, *and outputs a pair* (pk, sk) *consisting of a public key* pk *and a secret key* sk; *denoted:* $(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$.
- Enc *(encryption) is a randomized algorithm that takes as input a public key* pk *and a message* $m \in \mathcal{M}$, *and outputs a ciphertext* c; *denoted:* $c \leftarrow \mathsf{Enc}_{pk}(m)$.
- Dec *(decryption) is a deterministic algorithm that takes as input a secret key* sk *and a ciphertext* c, *and outputs a decrypted message* m'; *denoted:* $m' \leftarrow \mathsf{Dec}_{sk}(c)$.
- Eval *(homomorphic evaluation) takes as input the public key* pk, *a circuit* $C\colon \mathcal{M}^\ell \to \mathcal{M}$, *and ciphertexts* $c_1, \ldots, c_\ell$, *and outputs a ciphertext* $\widehat{c}$; *denoted:* $\widehat{c} \leftarrow \mathsf{Eval}_{pk}(C; c_1, \ldots, c_\ell)$.

The correctness *requirement is that for every* (pk, sk) *in the range of* $\mathsf{Gen}(1^\lambda)$ *and every message* $m \in \mathcal{M}$,

$$\Pr[\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(m)) = m] \geq 1 - \mathsf{neg}(\lambda);$$

the scheme is $\mathcal{C}$-homomorphic *for a circuit family* $\mathcal{C}$ *if for all* $C \in \mathcal{C}$ *and for all inputs* $x_1, \ldots, x_\ell$ *to* C, *letting* $(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$ *and* $c_i \leftarrow \mathsf{Enc}_{pk}(x_i)$ *it holds that:*

$$\Pr[\mathsf{Dec}_{sk}(\mathsf{Eval}_{pk}(C; c_1, \ldots, c_\ell)) \neq C(x_1, \ldots, x_\ell)] \leq \mathsf{neg}(\lambda)$$

(the probability is over all randomness in the experiment).

[2] Runtime were measured on comparable hardware: Intel Xeon Silver 4210 CPU2.20GHz (single core) in [47], and Intel Xeon Platinum 8269CY CPU2.50GHz (20-cores) in [49], both are comparable to AWS EC2 M5.2xlarge used in our experiments (according to the public benchmark [2]).

A $\mathcal{C}$-homomorphic encryption scheme supports *additive homomorphism over the reals* if $\mathcal{C}$ contains the circuits consisting of addition gates over the reals.

A HE scheme $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ is *CPA-secure* if no ppt adversary $\mathcal{A}$ can distinguish between the encryption of two equal length messages x_0, x_1 of his choice. See a formal definition in [41].

Psedudorandom functions (PRF). We call an efficiently computable family of keyed functions $\mathcal{F} = \{f_k \colon \{0,1\}^* \to \mathrm{B}\}_{k \in \{0,1\}^\lambda, \lambda \in \mathbb{N}}$ *pseudorandom* if for all λ, a uniformly random function f_k from $\mathcal{F}$ s.t. $|k| = \lambda$ is computationally indistinguishable from a uniformly random function from the set of all functions having the same domain and range. See a formal definition in [41, Definition 3.25].

3 Secret Sharing over Reals

We present our 2-out-of-2 secret sharing scheme over the reals (cf. an overview in Section 1.2). We start by discussing abstract real numbers (Figure 2), and then elaborate how to implement it for floating-point numbers in finite precision (Figure 3). Our scheme satisfies the following properties (cf. Theorem 3.2-3.3).

Definition 3.1 (secret sharing properties)**.** *Let $\mathcal{S} = (\mathsf{Shr}, \mathsf{Rec})$ be a 2-out-of-2 secret sharing scheme for A. Denote the set of possible values for share $i = 1,2$, by $\mathcal{S}_i = \{s_i \mid x \in \mathsf{A}, (s_1,s_2) \leftarrow \mathsf{Shr}(x)\}$. For every $s_1 \in \mathcal{S}_1$, denote by Shr_{s_1} and Rec_{s_1} the algorithms Shr and Rec respectively with the 1st share hardwired to be s_1. Let $\mathcal{E}$ be a $\mathcal{C}$-homomorphic public-key encryption scheme.*

- *$\mathcal{S}$ is* $\mathcal{E}$-friendly *if for all $s_1 \in \mathcal{S}_1$, $\mathsf{Rec}_{s_1} \in \mathcal{C}$ (and the input to Rec_{s_1} is in the message space of $\mathcal{E}$).*
- *$\mathcal{S}$ has a* random 1st share *if for all $x \in \mathsf{A}$,*

$$\{(s_1,s_2) \leftarrow \mathsf{Shr}(x)\} \equiv \{(s_1,s_2) \mid s_1 \leftarrow_R \mathcal{S}_1, s_2 \leftarrow \mathsf{Shr}_{s_1}(x)\}.$$

- *$\mathcal{S}$ has a* compact 2nd share *if for all $x \in \mathsf{A}$ and $s_1 \in \mathcal{S}_1$,*

$$|\mathsf{Shr}_{s_1}(x)| = |x|$$

(where $|z|$ denotes the binary representation length of z)

3.1 Abstract Real Numbers

We present our secret sharing scheme for $[0,1]$ in Figure 2. We prove that it satisfies the following properties.

Theorem 3.2 (secret sharing)**.** *The secret sharing scheme $\mathcal{S} = (\mathsf{Shr}, \mathsf{Rec})$ in Figure 2 satisfies the following:*

- *$\mathcal{S}$ is a 2-out-2 secret sharing scheme for $\mathsf{A} = [0,1]$.*
- *$\mathcal{S}$ is $\mathcal{E}$-friendly w.r.t. every HE scheme $\mathcal{E}$ that supports additive homomorphism over the reals (and whose message space contains $\mathcal{S}_2$).*
- *$\mathcal{S}$ has a random 1st share and a compact 2nd share.*

$\mathsf{Shr}(x)$. Given $x \in [0,1]$, proceed as follows:

1. Sample uniformly random $(b,t) \in \{0,1\} \times [0,1)$
2. Compute:
$$s_{int} = (-1)^b (\lfloor x+t \rfloor - b)$$
$$s_{frac} = x + t - \lfloor x+t \rfloor$$
3. Output $s_1 = (b,t)$ and $s_2 = (s_{int}, s_{frac})$

$\mathsf{Rec}(s_1,s_2)$. Given shares $s_1 = (b,t)$ and $s_2 = (s_{int}, s_{frac})$ in $\{0,1\} \times [0,1)$, output: $s_{frac} - t + (-1)^b s_{int} + b$

Figure 2: Our secret sharing (all arithmetic is over the reals)

Proof of Theorem 3.2, correctness. To prove correctness holds we show that for all $x \in \mathsf{A}$, $\mathsf{Rec}(\mathsf{Shr}(x)) = x$. Fix $x \in [0,1]$ and let $(s_1,s_2) \leftarrow \mathsf{Shr}(x)$. By definition of Shr there exists $b \in \{0,1\}$ and $t \in [0,1)$ such that $s_1 = (b,t)$ and $s_2 = (s_{int}, s_{frac})$ for $s_{int} = (-1)^b (\lfloor x+t \rfloor - b)$ and $s_{frac} = x + t - \lfloor x+t \rfloor$. Rec computes $x' = s_{frac} - t + (-1)^b s_{int} + b$. Assigning the value $s_{int} = (-1)^b (\lfloor x+t \rfloor - b)$ we have that:

$$\begin{aligned} x' &= s_{frac} - t + (-1)^b ((-1)^b (\lfloor x+t \rfloor - b)) + b \\ &= s_{frac} - t + (\lfloor x+t \rfloor - b) + b \end{aligned}$$

Assigning the value $s_{frac} = x + t - \lfloor x+t \rfloor$ we have that:

$$x' = x + t - \lfloor x+t \rfloor - t + (\lfloor x+t \rfloor - b) + b$$

which is in turn equal to x, and hence correctness holds. □

Proof of Theorem 3.2, security. To prove that (perfect) security holds we show that for every $x \in \mathsf{A}$ and $i \in \{1,2\}$, the distribution $\{s_i\}_{(s_1,s_2) \leftarrow \mathsf{Shr}(x)}$ is uniformly random in $\{0,1\} \times [0,1)$. The first share s_1 is uniformly random in $\{0,1\} \times [0,1)$ by construction. We next show that the pair $s_2 = (s_{int}, s_{frac})$ is uniformly random in $\{0,1\} \times [0,1)$. The integral part $s_{int} = (-1)^b (\lfloor x+t \rfloor - b)$ is the XOR of $\lfloor x+t \rfloor$ with a uniformly random bit b, and therefore s_{int} is a uniformly random bit. The fractional part $s_{frac} = x + t - \lfloor x+t \rfloor$ is equal to $(x+t)$ mod 1 for a uniformly random $t \in [0,1)$, and therefore s_{frac} is uniformly random in $[0,1)$. Moreover, since the randomness in s_{int} and s_{frac} emanates from the independent random variables b and t respectively, then s_{int} and s_{frac} are likewise independent, and so the pair (s_{int}, s_{frac}) is uniformly random in $\{0,1\} \times [0,1)$. We conclude that s_1 and s_2 are uniformly random in $\{0,1\} \times [0,1)$. This in turn

implies that for every $x, x' \in \mathsf{A}$ and $i \in \{1,2\}$, the two distributions $\{s_i\}_{(s_1,s_2)\leftarrow \mathsf{Shr}(x)}$ and $\{s'_i\}_{(s'_1,s'_2)\leftarrow \mathsf{Shr}(x')}$ are identical, and therefore perfect security holds. □

Proof of Theorem 3.2, HE-friendliness over the reals. The reconstruction algorithm $\mathsf{Rec}_{(b,t)}(s_{int}, s_{frac})$ computes over the reals the degree 1 polynomial $s_{frac} + \alpha s_{int} + \beta$, where the coefficients are $\alpha = (-1)^b$ and $\beta = b - t$, and the unknowns are s_{int} and s_{frac}. □

Proof of Theorem 3.2, random 1st share & compact 2nd share. For every $x \in [0,1]$ and $(s_1, s_2) \leftarrow \mathsf{Shr}(x)$, s_1 is uniformly random in $\{0,1\} \times [0,1)$, and $s_2 = \mathsf{Shr}_{s_1}(x)$ (by definition of $\mathsf{Shr}_{s_1}(x)$). Moreover, $|s_2| = |x|$ (when x and s_2 are specified with the same precision). □

3.2 Reals in Floating-Point Representation

We explain now how to apply our scheme on inputs $x \in [0,1]$ specified in floating point representation. The scheme is parameterized by a precision p, and is denoted by $(\mathsf{Shr}_p, \mathsf{Rec})$. The algorithm Shr_p first converts the input x to fixed-point in $Q1.p$ format (i.e., with 1 bit for the integral part and p bits for the fractional part); then proceeds analogously to Figure 2, albeit while sampling t in $Q1.p$ format; finally, outputs the shares after conversion to floating point. The algorithm Rec is identical to Figure 2, albeit computing on floating-point numbers. The latter is done for compatibility with the message space in libraries implementing CKKS encryption [23], such as Microsoft SEAL [57], where messages are floating point numbers in double precision.

More formally, given $p \in \mathbb{N}$, for all $x \in [0,1]$, denote by x_p the value of x in fixed-point representation $Q1.p$ format (possibly losing accuracy in the conversion to $Q1.p$). We say that $(\mathsf{Shr}, \mathsf{Rec})$ is a *secret sharing scheme with precision p for* $[0,1]$ if it is secure (as defined in 2.1) and correct w.r.t x_p in the following sense: for all $x \in [0,1]$, $\mathsf{Rec}(\mathsf{Shr}_p(x)) = x_p$. The scheme is specified in Figure 3 and analyzed in Theorem 3.3.

Theorem 3.3 (secret sharing, finite precision)**.** *For every $p \in \mathbb{N}$, the secret sharing scheme $\mathcal{S} = (\mathsf{Shr}_p, \mathsf{Rec})$ in Figure 3 satisfies the following.*

- *$\mathcal{S}$ is a 2-out-of-2 secret sharing scheme with precision p for $[0,1]$.*
- *$\mathcal{S}$ is $\mathcal{E}$-friendly w.r.t. every HE scheme $\mathcal{E}$ that supports additive homomorphism over the reals (and whose message space contains $\mathcal{S}_2$).*[3]
- *$\mathcal{S}$ has a random 1st share. Moreover, $\mathcal{S}$ has a compact 2nd share, provided that the input x is specified with significand and exponent precision at least p and $\log_2 p$, respectively.*[4]

[3] We may convert $\mathcal{S}_2$ to the message space of $\mathcal{E}$, if needed.

[4] For x in double precision, taking $p = 53$ guarantees that the scheme has a compact 2nd share.

$\mathsf{Shr}_p(x)$. Given $x \in [0,1]$ in finite precision, specified in a floating point representation, proceed as follows.

1. Convert x to binary fixed-point number in $Q1.p$ format (i.e., with 1 bit for the integral part and p bits for the fractional part, possibly losing precision by truncating the least significant bits)
2. Sample uniformly random $(b,t) \in \{0,1\} \times [0,1)$ where t is a binary fixed-point number in $Q1.p$ format
3. Compute:
$$s_{int} = (-1)^b (\lfloor x + t \rfloor - b)$$
$$s_{frac} = (x + t) - \lfloor x + t \rfloor$$
4. Output $s_1 = (b,t)$ and $s_2 = (s_{int}, s_{frac})$ with t and s_{frac} in a floating point format that preserves their value[a]

[a] For example, floating point in *double* precision suffices for $p = 53$.

$\mathsf{Rec}(s_1, s_2)$. Given shares $s_1 = (b,t)$ and $s_2 = (s_{int}, s_{frac})$ in $\{0,1\} \times [0,1)$ (in the same format as in the output of Shr), output:
$$s_{frac} - t + (-1)^b s_{int} + b$$

Figure 3: Our secret sharing in finite precision p

Proof of Theorem 3.3, correctness. Given $p \in \mathbb{N}$ and $x \in [0,1]$ in floating point representation, we show that $\mathsf{Rec}(\mathsf{Shr}_p(x)) = x_p$. Since the conversion to floating point in Step 4 of Figure 3 preserves the values of t and s_{frac}, it follows that s_1 and s_2 inputted to Rec satisfy: $s_{frac} = x_p + t - \lfloor x_p + t \rfloor$ and $(-1)^b s_{int} + b = \lfloor x_p + t \rfloor$ Therefore the output of Rec satisfies, $s_{frac} - t + (-1)^b s_{int} + b = x_p$ as desired. □

Proof of Theorem 3.3, security. Fix $p \in \mathbb{N}$. Consider first the algorithm Shr^*_p that is the same as Shr_p except for not converting t and s_{frac} to floating point in Step 4. Like in Theorem 3.2, b and s_{int} are uniformly random in $\{0,1\}$. Denote by $[0,1)_p$ the set of numbers from $[0,1)$ in $Q1.p$ format. In Step 2, t is sampled uniformly at random from $[0,1)_p$. Moreover, in Step 3, $s_{frac} = x_p + t \bmod 1$, so it is likewise uniformly random in $[0,1)_p$. Moreover, b and t are sampled independently, and so also s_{int} and s_{frac} are statistically independent. Therefore, (b,t) and (s_{int}, s_{frac}) are both distributed uniformly at random in $\{0,1\} \times [0,1)_p$. This implies that for every $p \in \mathbb{N}$, $x, x' \in [0,1]$ and $i \in \{1,2\}$, the distributions $\{s_i\}_{(s_1,s_2)\leftarrow \mathsf{Shr}^*_p(x)}$ and $\{s'_i\}_{(s'_1,s'_2)\leftarrow \mathsf{Shr}^*_p(x')}$ are identical. Now consider the algorithm Shr where we do convert t and s_{frac} to floating-point representation. Because the conversion to floating-point does not increase the statistical distance, the distributions $\{s_i\}_{(s_1,s_2)\leftarrow \mathsf{Shr}_p(x)}$ and $\{s'_i\}_{(s'_1,s'_2)\leftarrow \mathsf{Shr}_p(x')}$ are identical. Namely, perfect security holds. □

Proof of Theorem 3.3, $\mathcal{E}$-friendly and random 1st share. The proof follows identically to the proof of Theorem 3.2. □

Proof of Theorem 3.3, compact 2nd share. To convert $Q1.p$ to floating-point with no information loss it suffices to use a floating-point format with significand and exponent of precision p and $\log_2 p$ respectively. Concretely, we can convert $z = (z_0, \ldots, z_{p-1})$ to the floating point number with significand z and exponent $-p$ (in binary representation). □

4 Compact Storage for HE

In this section we present our compact storage for HE-retrieval (Section 4.1 and Figure 4) and discuss how our solution fits in within the larger system (Section 4.2). See also an overview of our solution in Section 1.2, an illustration in Figure 1, and a system flow diagram in Figure 5. We provide a rigorous security analysis in Section 4.3.

4.1 Our Protocol

Our compact storage with HE-retrieval (Figure 4) is comprised of two protocols: store and retrieve. In store, the data producer dataProd secret shares the data and uploads it to storage; in retrieve, the two servers compSrv and auxService run a two-party protocol, at the conclusion of which, compSrv holds a HE ciphertext for the retrieved data. The system relies on a trusted key management service (KMS) for the trusted setup phase in Figure 4 (cf. Section 4.2).

Our solution is generic and can be instantiated with any HE-scheme $\mathcal{E}$ and secret sharing scheme $\mathcal{S}$, provided that $\mathcal{S}$ is $\mathcal{E}$-friendly and has a random 1st share and compact 2nd share (cf. Definition 3.1). For example, it can be instantiated with additive secret sharing over $\mathbb{Z}_n$ together with any HE that supports additive homomorphism over that ring, e.g., BGV [17].[5] In the context of our motivating scenario, our compact storage solution can be instantiated with our secret sharing scheme for secrets in $\mathsf{A} = [0,1]$ (cf. Figures 2-3), together with any HE scheme that supports additive homomorphism over the reals, e.g., CKKS [23].

4.2 System Architecture

We now describe how our solution is executed within the larger system, including applications requesting the computation, a persistent storage resource, and a key management service (KMS).

The system flow. The retrieve protocol is initiated by a compute request sent from an application (with identifier appID) to compSrv, specifying the data location *index* to compute on.

[5] In additive secret sharing for $\mathbb{Z}_n$, $\mathsf{Shr}(x)$ outputs uniformly random $s_1 \in \mathbb{Z}_n$ and $s_2 = x + s_1 \bmod n$ and $\mathsf{Rec}(s_1, s_2)$ outputs $s_1 + s_2 \bmod n$. Clearly, this has a random 1st share, compact 2nd share, and is friendly with respect to any additive-homomorphic encryption over $\mathbb{Z}_n$.

Common parameters: A $\mathcal{C}$-homomorphic CPA-secure HE scheme $\mathcal{E} = (\mathsf{HE.Gen}, \mathsf{HE.Enc}, \mathsf{HE.Dec}, \mathsf{HE.Eval})$. A 2-out-of-2 secret sharing scheme $\mathcal{S} = (\mathsf{Shr}, \mathsf{Rec})$ for a domain A s.t. $\mathcal{S}$ is $\mathcal{E}$-friendly, has a random 1st share and compact 2nd share, and $(\mathsf{Shr}_{s_1}, \mathsf{Rec}_{s_1})$ as specified in Definition 3.1. A pseudorandom family of functions $\mathcal{F} = \{f_k \colon \{0,1\}^* \to \mathcal{S}_1\}_{k \in \{0,1\}^\lambda}$. A security parameter λ.

Parties: dataProd, compSrv and auxService.

Trusted setup: Sample $(pk, sk) \leftarrow \mathsf{HE.Gen}(1^\lambda)$ and $k \leftarrow \{0,1\}^\lambda$. Give pk to compSrv and auxService; and f_k to dataProd and compSrv.

Storage: store is executed by dataProd on input $(index, x) \in \{0,1\}^* \times \mathsf{A}$, as follows.

1. Compute $s_1 \leftarrow f_k(index)$ and $s_2 \leftarrow \mathsf{Shr}_{s_1}(x)$
2. Upload to storage $(index, s_2)$ with access authorized to auxService

Retrieval: retrieve is executed by compSrv and auxService, where compSrv has input *index* (auxService has no input):

1. compSrv computes $s_1 \leftarrow f_k(index)$, and sends *index* to auxService
2. auxService does the following:
   (a) Download from storage $(index, s_2)$
   (b) Compute $c_2 \leftarrow \mathsf{HE.Enc}_{pk}(s_2)$ (if s_2 is a tuple, encrypt entry-by-entry)
   (c) Send c_2 to compSrv
3. compSrv outputs $c \leftarrow \mathsf{Eval}_{pk}(\mathsf{Rec}_{s_1}; c_2)$

Figure 4: Compact storage for HE over domain A. For example, with $\mathcal{S}$ of Figure 2, $s_1 = (b,t)$, $s_2 = (s_{int}, s_{frac})$, and $c_2 = (\mathsf{HE.Enc}_{pk}(s_{int}), \mathsf{HE.Enc}_{pk}(s_{frac}))$.

Upon receiving such a request, compSrv proceeds as follows: (1) fetch from the KMS the PRF key and the HE public key associated with the data at *index*; (2) forward (*index*, appID) to auxService; (3) upon receiving from auxService a HE encrypted share, homomorphically reconstruct the data; (4) perform the requested computation over the HE encrypted data. Upon receiving a request from compSrv, auxService proceeds as follows: (1) fetch from the KMS the HE public key associated with data at *index*; (2) download from storage the share $s_2(index)$ in *index* location, and encrypt it using the HE public key; (3) send the HE-encrypted share to compSrv. The KMS handles fetch requests according to the access policy, returning DENY in case the application appID is not authorized to compute on data at location *index*. We note that appID

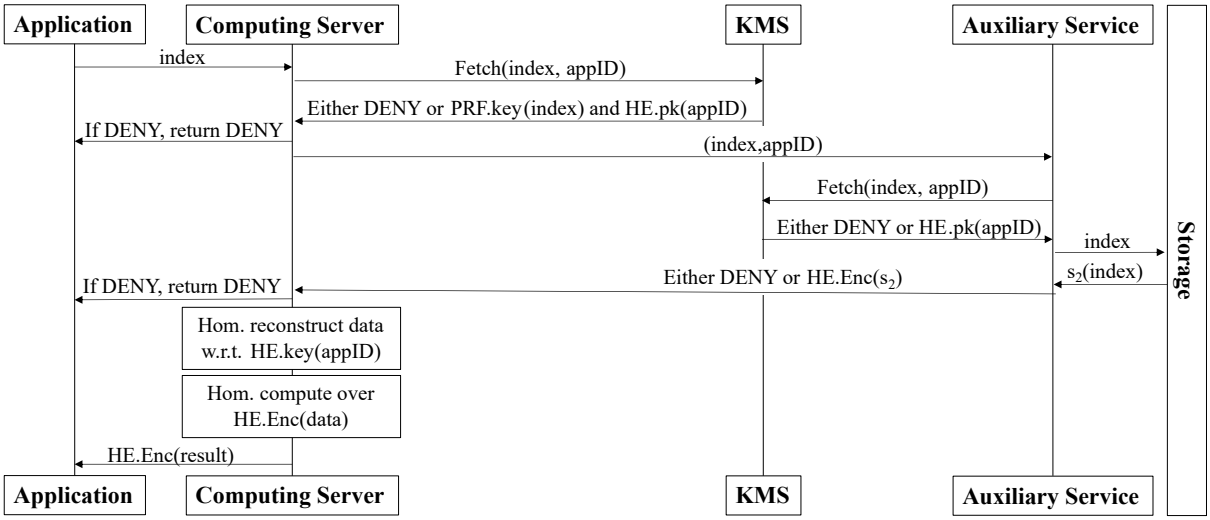


Figure 5: System architecture and flow

is included in the fetch request, to enable verifying that the application is authorized to compute on the data at location *index*. The storage handles download requests according to the access policy, returning DENY in case the requesting entity is not authorized to download the data at location *index*. See Figure 5 for the described flow. For simplicity, we presented the flow for a single index and HE key. More generally, the application can specify a vector of indices together with a batching pattern and a HE key identifier (indicating under which public key the retrieval and computation should be executed). Details follow.

Batched retrieval. The retrieval protocol retrieve can retrieve multiple data items specified by their indices together with (optionally) a specification of which data items to pack together in each retrieved HE ciphertext (batching pattern). Importantly, this batched retrieval supports dynamic and flexible choice of which data items to pack together in each ciphertext of the HE (while complying, of course, with the upper bound on how many data items can be packed in each ciphertext). To achieve batched retrieval, when auxService encrypts its shares, it packs in each HE ciphertext the shares corresponding to the requested data items. Likewise, the computing server encodes its plaintext shares while adhering to the requested batching pattern, and upon receiving the packed encrypted shares from auxService homomorphically evaluates the reconstruction algorithm in a single-instruction-multiple-data (SIMD) fashion.

Managing authorizations and keys. To enforce access and authorization policies, each system entity has a unique identifier, where we use standard mechanisms and services for registering and authenticating identities (with proper credentials), and for managing the associated keys and policies. For simplicity of the presentation we focus on access control granularity at the data producer level, where for each data producer, an application is either fully authorized (i.e., it is authorized to request computations on all data uploaded by the data producer), or the application is fully denied (i.e., it is authorized to none of the data producer's data).

The KMS maintains for each data producer the list of authorized identities as detailed next. During an on-boarding phase, each data producer is on-boarded to the KMS, and the KMS generates for it a PRF key, and a HE key pair that is used for homomorphic computations on this data producer's data (or, more generally, several key pairs, for different schemes and parameters, to support versatile homomorphic computations). Each of the generated HE keys is associated with an access policy, i.e., a list of identities specifying which applications are authorized to request computations on the data producer's data and to fetch the HE secret key to obtain the result in cleartext. Likewise, applications are on-boarded to the KMS by adding them to the access policies of the relevant data producers keys. Subsequently, the public key of a data producer can be fetched from the KMS, by any entity, by specifying an application's identity (appID) that is in the list of applications authorized to compute on the data of this data producer; whereas the secret key can be fetched only by the authorized applications themselves. The PRF key, of each data producer, can be fetched only by compSrv.

Our system guarantees that, for each data producer dataProd, only authorized applications can obtain cleartext computation results on dataProd's data. Moreover, even if one of the servers (compSrv or auxService) is malicious, it cannot bias the HE retrieval to be performed under a public key other than the key associated with dataProd's data. Even an active attack by compSrv sending arbitrary $index^*$, results in obtaining the share at location $index^*$ encrypted under the HE public key of the data producer associated with

this location. Although compSrv can use the received ciphertext to homomorphically reconstruct the data, it cannot fetch the secret key and decrypt if it is not authorized to this data producer's data.

Formal modeling of key management and storage. We model the KMS and storage resources as ideal functionalities, denoted $\mathcal{F}_{\text{storage}}$ and $\mathcal{F}_{\text{keys}}$, respectively. These functionalities may be accessed by all system entities (including ones beyond those considered in our protocol), using the same persistent resources in all sessions and all protocols running in the ecosystem.

The key management functionality $\mathcal{F}_{\text{keys}}$ (Figure 7) is parameterized by a data producer dataProd, a HE scheme $\mathcal{E}$, and a set of indices *ids* specifying the entities that are authorized to access the secret key. We remark that, in Figure 5, fetch requests include also *index*, as an extra parameter; this should be interpreted as follows. Upon receiving a fetch request for $(index, \text{appID})$, first identify the data producer dataProd that is associated with the data at *index* (assuming that, as common in industrial practices, the metadata at location *index* includes the identity of dataProd who uploaded the data), and then send $(\text{Fetch}, \text{appID})$ request to the functionality $\mathcal{F}_{\text{keys}}$ that is parameterized on this dataProd. Moreover, to support multiple HE schemes and keys, $\mathcal{F}_{\text{keys}}$ initializes the keys, stores them with unique key identifiers (keyID's). Then fetch requests include appID together with keyID.

The storage functionality $\mathcal{F}_{\text{storage}}$ (Figure 6) maintains uploaded records of the form $(index, value, \text{IDs})$, where *index* is a unique record identifier, *value* is the stored data, and IDs is the set of parties authorized to download the record. Download requests are answered only if the requesting entity is authorized to receive the data, i.e., the entity appears in the IDs of the requested *index*.

Our protocol (Figure 4) is realized using the ideal functionalities $\mathcal{F}_{\text{storage}}$ and $\mathcal{F}_{\text{keys}}$ as follows. The trusted setup is executed by calling $\mathcal{F}_{\text{keys}}$ initialization to generate the HE and PRF keys. In retrieve, both compSrv and auxService send $(\text{Fetch}, \text{appID})$ to $\mathcal{F}_{\text{keys}}$, and receive in response the HE public key (provided appID is authorized); in addition, compSrv receives also the PRF key. Upload and download operations (in store and retrieve, respectively) are executed by sending to $\mathcal{F}_{\text{storage}}$ $(\text{Upload}, index, s_2, \text{auxService})$ and $(\text{Download}, index)$ requests, respectively.

4.3 Security Analysis

Our solution is compact, private, and correct in the following sense. *Compactness* is in the sense that there is no overhead in storage size over standard storage such as storing AES encrypted data. *Privacy* is in the sense that data secrecy is preserved at all time, against any active adversary controlling at most one out of the servers compSrv and auxService. *Correctness* is in the sense that the retrieved HE ciphertext decrypts to the stored value (and all parties are ppt). That

Shared Functionality $\mathcal{F}_{\text{storage}}$

Shared functionality $\mathcal{F}_{\text{storage}}$ proceeds as follows:

Upload. Upon receiving from a party P a message:

$$(\text{Upload}, index, value, \text{IDs})$$

record the tuple $(index, value, \text{IDs})$ (ignore if a record $(index, \cdot, \cdot)$ was previously recorded).

Download. Upon receiving from a party P_j a message:

$$(\text{Download}, index)$$

output $(index, value)$ to P_j, if $(index, value, \text{IDs})$ was previously recorded by $\mathcal{F}_{\text{storage}}$ and IDs contains P_j (Otherwise return DENY).

Figure 6: The storage functionality

Shared Functionality $\mathcal{F}_{\text{keys}}$

Parameterized by data producer dataProd, a HE scheme $\mathcal{E} = (\text{Gen}, \text{Enc}, \text{Dec}, \text{Eval})$, and identities *ids*, shared functionality $\mathcal{F}_{\text{keys}}$ proceeds as follows:

Init. Sample and record $(pk, sk) \leftarrow \text{Gen}(1^\lambda)$ and $k \leftarrow \{0,1\}^\lambda$. Output k to dataProd.

Fetch key. Upon receiving a $(\text{Fetch}, \text{appID})$ message from a party P proceed as follows:

- If $P \in ids$ return (pk, sk)
- If appID $\in ids$ return pk and if P is compSrv then additionally return k
- Otherwise return DENY

Figure 7: The keys functionality

is, the output ciphertext c obtained by first executing store on data item x and database location *index* and then executing retrieve on *index* satisfies that $\text{HE.Dec}_{sk}(c) = x$ (where sk is the secret key for HE as generated during setup). To formally state these properties we first set some notations. We denote the output and view of compSrv in an execution of the protocol in Figure 4 by

$$\text{out}_{\text{compSrv}}(x; index, \lambda) \text{ and}$$
$$\text{view}_{\text{compSrv}}(x; index, \lambda)$$

respectively, where the view consists of the party's input, randomness and received messages. Analogously, we denote the view of auxService by $\text{view}_{\text{auxService}}(x; index, \lambda)$. We denote

the values uploaded to storage during the execution by

$$\mathsf{storage}(x; index, \lambda)$$

and denote their storage size by $|\mathsf{storage}(x; index, \lambda)|$. We compare the latter to the size $|x|$ of directly storing x.

Theorem 4.1 (compact storage for HE over domain A)**.** *The protocol in Figure 4 satisfies the following:*

- Compactness. *For every* $(index, x) \in \{0,1\}^* \times \mathsf{A}$ *and* $\lambda \in \mathbb{N}$,
$$|\mathsf{storage}(x; index, \lambda)| = |x|$$
- Privacy *holds against any (active)* ppt *adversary controlling either* compSrv *or* auxService*, but not both. Formally, for every* ppt $P^* \in \{\mathsf{compSrv}^*, \mathsf{auxService}^*\}$*, and every* ppt *distinguisher* $\mathcal{D}$ *that chooses index and* $x_0, x_1 \in \mathsf{A}$ *s.t.* $|x_0| = |x_1|$,
$$\Big|\Pr[\mathcal{D}(\mathsf{view}_{P^*}(x_0; index, \lambda)) = 1] - \Pr[\mathcal{D}(\mathsf{view}_{P^*}(x_1; index, \lambda)) = 1]\Big| \leq \mathsf{neg}(\lambda)$$
(the probability is over all randomness in the protocol).
- Correctness.[6] *For every* $(index, x) \in \{0,1\}^* \times \mathsf{A}$*, with probability at least* $1 - \mathsf{neg}(\lambda)$ *the following holds:*
$$\mathsf{Dec}_{sk}\left(\mathsf{out}_{\mathsf{compSrv}}(x; index, \lambda)\right) = x$$
(the probability is over all randomness in the protocol). Moreover, dataProd*,* compSrv *and* auxService *are* ppt*.*

Proof. The proof appears in Appendix A. □

5 Empirical Evaluation

We implemented our compact storage with HE-retrieval (Figure 4) into a system, named CSHER, and ran extensive experiments demonstrating that our system has compact storage with zero overhead over storing AES encrypted data, and nearly optimal runtime complexity.

5.1 The Implemented System

CSHER is instantiated with: (1) CKKS [23] HE scheme using Microsoft SEAL version 3.6.2 [57]; (2) our secret sharing scheme for reals (Figure 2); and (3) a PRF based on HMAC with SHA-256 using OpenSSL version 1.1.1. The implementation is in C++, compiled with g++ version 9.3.0 and C++ standard 17. Everything is running on a Docker container, based on an Ubuntu 20.04.2 image. Access control is implemented by using AES-GCM-256 symmetric key encryption to encrypt stored shares for the data under a secret-key accessible to dataProd and auxService, but not to compSrv. As an optimization we upload to storage the AES encryption of $s'_2 = s_{int} + s_{frac}$, so that only a single ciphertext is required for each data item; when auxService downloads the AES ciphertext of s'_2, it decrypts and computes $s_2 = (\lfloor s'_2 \rfloor, s'_2 - \lfloor s'_2 \rfloor)$. We ran experiments in both a single-cloud and a multi-cloud environment.

Single cloud system. We ran our implementation on AWS EC2 instances with AWS S3 storage. We use a separate AWS EC2 instance for each entity (dataProd, compSrv and auxService), all residing in the same AWS subnet. Our implementation is single threaded and runs on standard EC2 instances of type M5.2xlarge (32 GB RAM and up to 10 Gbps network bandwidth). The storage is in an AWS S3 bucket residing in the same AWS region as the EC2 instances. Communication between compSrv and auxService is via HTTP using Microsoft's cpprestsdk library.

Multi cloud system. We ran our implementation in a multi-cloud environment where compSrv runs on a Google Cloud (GC) Virtual Machine (VM) of type n2-standard-8, while dataProd, auxService and the storage are identical to those used in our single-cloud system. We note that EC2 m5.2xlarge and GC VM n2-standard-8 instances are similar in characteristics, both having 8 vCPUs and 32GB memory, but not necessarily identical. Our Google Cloud Virtual Machine and the AWS EC2 instances reside in GC us-west1 and AWS us-west-2, respectively.

5.2 Experiments

We measured storage size, runtime and communication in both our single-cloud and multi-cloud systems and compared performance to the baseline solution of storing and downloading the data directly in HE-encrypted form using Zstandard library compression. Furthermore, we report performance of a full pipeline execution consisting of HE-retrieval followed by homomorphic computation over the HE-retrieved data.

We use the same parameters when executing both our system and the baseline solution, as follows. Both are implemented with CKKS in Microsoft SEAL as the HE scheme, using the same AWS S3 bucket for storage, and with security parameter 128 bits. The degree of the cyclotomic polynomial ranged over all values supported in SEAL: 8192, 16384 and 32768, with 30 bits of precision for the plaintext moduli at each level of homomorphic computation. The number slots of data items that can be packed in each ciphertext is half the degree of the cyclotomic polynomial. Performance is evaluated both in batched mode (packing slots data items in each ciphertext)[7] and in un-batched mode (a single data item in

[6] If the protocol employs a secret sharing scheme for $[0,1]$ with precision p (cf. Section 3.2), then correctness holds w.r.t x_p, i.e., $\mathsf{Dec}_{sk}\left(\mathsf{out}_{\mathsf{compSrv}}(x; index, \lambda)\right) = x_p$ (where x_p is the value returned when converting x to $Q1.p$ fixed-point format).

[7] This affects storage size not only for HE, but also for AES where batching

each ciphertext).

Data items are synthetically generated from $[0, 1]$ with finite precision, and represented as double-precision numbers according to the IEEE 754 standard [3]. The randomness in the secret sharing is taken as a fixed-point number with the same precision. The number of data items n ranges over the following values: 2, 4096, 16384, 98304, 507904 and 2031616.

Keys are kept in cache memory of the relevant party, and are reused across executions of the protocol. Each experiment is repeated 20 times, taking the average and standard-deviation.

We ran both end-to-end and microbenchmarks experiments. The end-to-end experiments compare the performance of our system vs. the baseline solution in: storage size; end-to-end runtime during store; and end-to-end runtime during retrieve. The microbenchmarks measure runtime in each computation step and the communication.

5.3 Results

Performance of our HE-retrieval system. The empirical evaluation of both our single-cloud and our multi-cloud systems demonstrates that they attain:

- **Compact storage size:** $10\times$ better than the baseline in batched mode, and $10^4\times$ better if un-batched.
- **Fast end-to-end runtime:** only twice the optimal time.

See Table 2 and Figures 8a-8b for our results in experiments with a degree 8192 cyclotomic polynomial. We note that the storage phase in our solution is faster than the baseline, but retrieval is slower. Essentially, our storage is faster because we store AES ciphertexts rather than encrypting and storing HE ciphertexts; whereas our retrieval is slower because it includes HE encryption, and HTTP serialization and deserialization of the HE ciphertexts (rather than fetching from storage previously encrypted HE ciphertexts). We remark that the HTTP serialization and deserialization (provided by SEAL library) consume roughly half of the retrieval time. Table 4 shows the breakdown of end-to-end storage and retrieval runtime measurements into specific operations and HTTP communication complexity; see Table 1 for the correspondence to Figure 4. The experiments with degrees 16384 and 32768 cyclotomic polynomial show essentially no change in performance, provided full batching capacity is utilized (because the increase in complexity per ciphertext is fully compensated by increase in the number of slots per ciphertext).

We note that there is almost no performance difference in our single vs. multi cloud systems. Concretely we can see in Table 4 that, despite hugely different sizes of data sent over HTTP, the communication time has grown by an almost constant factor of 0.1 to 0.23 seconds. This is expected as the distance between the servers affects the latency but not the throughput. As evident in Table 2, the overall effect of the additional latency on retrieval time is small and becomes relatively negligible as data size and processing time increase. In fact for 2,031,616 data items we even see a small improvement in the retrieval time, and this is probably due to some differences in the hardware between the VMs.

means that we use one IV for many data items.

Retrieve-then-Evaluate. In Table 3 we report performance when executing our HE-retrieval solution followed by homomorphic evaluation of a decision tree over the HE-retrieved data. We HE-retrieved 1,966,080 data items, corresponding to 327,680 feature vectors (samples) of 6 features each, and homomorphically evaluated over them a depth four decision tree, using the algorithm from [6]. The HE parameters tightly support the homomorphic computation at hand (degree 16K cyclotomic polynomial with ciphertext modulus that is a product of eight 30-bits primes and a 50-bits special prime), and we fully utilized the batching capacity (8192 data items per ciphertext). This is made possible by our solution's capability to dynamically select HE parameters and batching pattern at retrieval time. Our results exhibit *fast performance*: 0.57ms runtime per data sample for the full pipeline. This runtime is comprised of 0.04ms (7%) for the homomorphic reconstruction and 0.53ms (93%) for the homomorphic tree evaluation. We note that the former is a constant overhead, independent of the homomorphic computation at hand; e.g., when executing homomorphic training, the former still requires 0.04ms while the latter takes much longer (minutes to hours when using the algorithm from [6]).

We compare our performance to the baseline solution of directly storing HE ciphertexts, while focusing on scenarios A-D detailed next (cf. Table 3).

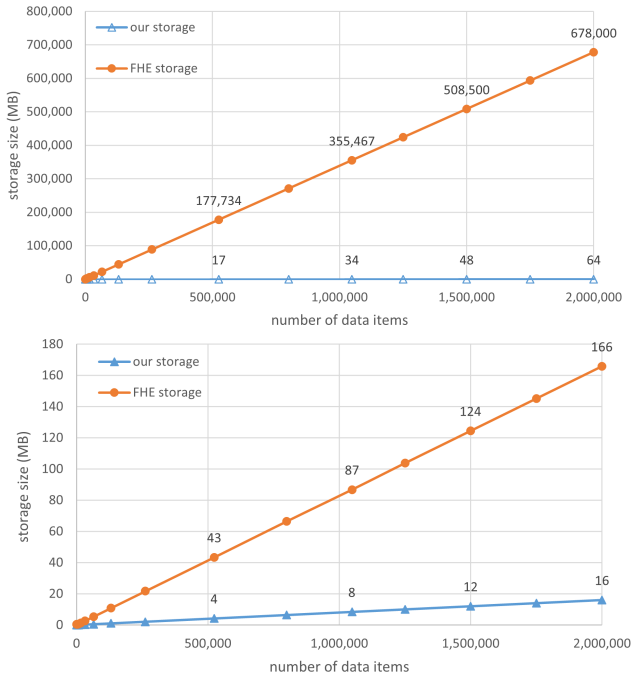
- *Scenario A:* the target computation and data are known at upload time. So HE parameters and batching profile are set as to optimize runtime (identically to the parameters in executing our system); but there is no support for homomorphic computations requiring higher HE parameters or different batching.
- *Scenario B:* target data and batching is known at upload time, but the target computation is not known. So, HE parameters are set to support versatile computations (degree 32K cyclotomic polynomial with ciphertext modulus that is a product of ten 45-bits primes and a 60-bits special prime), and retrieved ciphertexts utilize full batching capacity (16384 data items in each ciphertext).
- *Scenario C:* the converse of B – the target computation is known at upload time, but the target data is not known at upload time. So, HE parameters are set as in Scenario A, and retrieving m data items may require, in the worst case, retrieving m ciphertexts (more generally, between m/slots and m, depending on the fraction of slots per ciphertext containing relevant data items).

Table 1: Microbenchmarks to Figure 4 steps correspondence

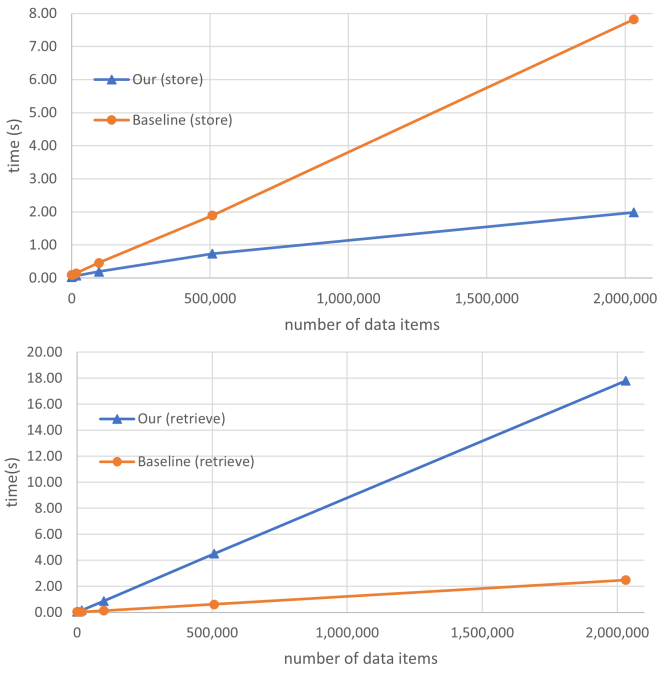
Entity	**Data Producer**			**Computing Server**			**Auxiliary Server**			
Benchmark operation	**Share**	**AES encrypt**	**Upload**	**Process HTTP response**	**HMAC share derivation**	**Reconstruct**	**Download**	**AES decrypt**	**HE encrypt**	**Prepare HTTP response**
Fig. 4 step # /other details	store1	store2	store2	read stream & deserialize	retrieve1	retrieve3	retrieve2a	retrieve2b	retrieve2b	serialize & write to stream

Table 2: Storage size (MB) and runtime (seconds) in end-to-end experiments

Number of data items	**Storage Size (MB)**				**Runtime (s)**				
	Batched		**Un-batched**		**Store**		**Retrieve**		
	Ours	**Baseline**	**Ours**	**Baseline**	**Ours**	**Baseline**	**Ours**		**Baseline**
							single cloud	**multi cloud**	
2	**0.00003**	0.3	**0.00006**	0.8	**0.02**	0.10	0.05	0.15	**0.03**
4,096	**0.03**	0.3	**0.1**	1,389.0	**0.06**	0.10	0.06	0.16	**0.04**
16,384	**0.10**	1.4	**0.5**	5,554.0	**0.07**	0.10	0.16	0.26	**0.05**
98,304	**0.80**	8.0	**3.0**	33,325.0	**0.20**	0.50	0.90	1.00	**0.10**
507,904	**4.00**	42.0	**16.0**	172,179.0	**0.70**	1.90	4.50	4.60	**0.60**
2,031,616	**16.00**	168.0	**65.0**	688,718.0	**2.00**	7.80	17.80	17.60	**2.50**



(a) Storage size (MB) in our solution vs. baseline, on data items encrypted one-by-one (top) or in batched form (bottom)



(b) End-to-end time (seconds) in our solution vs. baseline for storage (top) and retrieval (bottom)

Figure 8: Performance: storage size (left) and runtime (right)

Table 3: Retrieve-then-Evaluate Ours vs. Baseline A-D, on samples with 6 features.

	Versatile Homomorphic Computations?	Cherry Picking Data?	Storage per Sample (KB)	Runtime per Sample (ms)
A	×	×	0.50	**0.53**
B	✓	×	0.50	1.70
C	×	✓	3600.00	74.23
D	✓	✓	7200.00	158.39
Ours	✓	✓	**0.05**	0.57

- *Scenario D:* neither computation nor data are known at upload time. So, HE parameters are set as in Scenario B, and retrieving m data items is as in Scenario C.

In Scenarios C-D, not fully utilizing batching slows down the homomorphic computation on the retrieved ciphertexts to 28,827ms per sample; however, this may be remediated by first homomorphically packing the retrieved data, and then applying the homomorphic evaluation on the produced fully batched ciphertexts. In the empirical evaluation of Scenarios C-D, we implement homomorphic packing as follows: we store in each ciphertext a single data item replicated in all slots, and homomorphically pack the desired batch by homomorphically zeroing out unwanted positions and summing up the ciphertexts of interest.[8] This entails storing six ciphertexts for each sample (one for each feature); each ciphertext size is 0.6MB and 1.2MB in Scenarios C and D respectively. We note that it is possible to improve on the storage size while incurring a degradation in runtime performance. Concretely, when implementing Scenarios C-D, we could store fully batched ciphertexts (without replications), and homomorphically repack by employing rotations (together with zeroing out unwanted positions). However, this incurs more than 20× slowdown compared to the former approach, resulting in runtime over 3406ms per sample, with HE parameters of Scenario D. Moreover, the use of rotations incurs a higher noise accumulation, resulting in data corruption in Scenario C (due to its lower HE parameters).

Comparison to Baselines B-D. Our solution strictly outperforms Baselines B-D, in both storage and runtime. Our solution offers saving in storage by a factor of 10×, 72,000× and 144,000× compared to Scenarios B, C and D, respectively; and runtime speedup by a factor of 3×, 130× and 278×, respectively. See Table 3.[9]

[8] I.e., let x_j be the desired item for position j, $\mathbf{x}_j = (x_j, \ldots, x_j)$, and $\mathbf{e}_j$ the unit vector with 1 at position j, we homomorphically evaluate $\sum_{j=1}^{\text{slots}} \mathbf{e}_j \mathbf{x}_j$.

[9] Storage size per sample is the same in Scenarios A-B (cf. Table 3), despite B having higher HE parameters leading to larger ciphertexts. This is because Scenarios A-B store batched ciphertexts, and so the number of slots per ciphertexts increases proportionally to the increase in ciphertext size. In contrast, Scenarios C-D do not employ batching, and therefore the storage per sample increases with the increase in HE parameters.

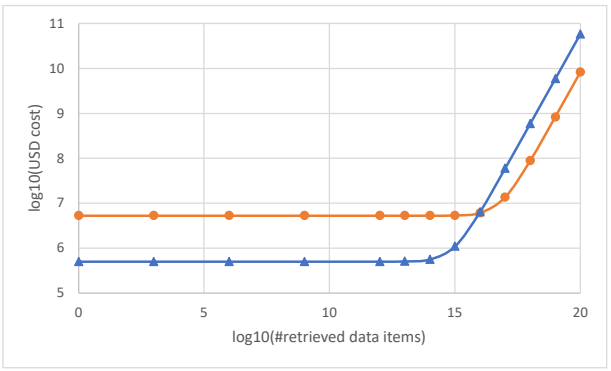


Figure 9: Costs in USD (y-axis) of storage and retrieval, based on AWS pricing, for Our solution (blue triangles) vs. Baseline A (orange circles), when storing 25PB of data and retrieving up to 10^{20} data items (x-axis), in log-log scale.

Comparison to Baseline A. Our solution outperforms Baseline A in storage (10× saving) and in its support for versatile homomorphic computations, data cherry picking and dynamic batching, while incurring only a minor degradation in runtime (0.04ms overhead per sample). See Table 3. We illustrate (Figure 9) the monetary implications of using our solution compared to Baseline A, by extrapolating our empirical measurements to massive amounts of data. We rely on AWS pricing [1] for M5.2xlarge EC2 machine and S3 bucket as used in our system ($0.242 per computing hour, and $0.02 for 1GB storage per month). Consider a data lake of 25PB of data. AWS storage costs approximately $0.5M per month; so our solution saves roughly $4.5M per month in storage costs compared to the baseline solution of directly storing HE ciphertexts. Furthermore, even when accounting for the overhead in HE-retrieval compared to directly retrieving HE ciphertexts – which is, 15.1s runtime for two million HE-retrieved data items (cf. Table 2), costing $0.001 – our solution is cost effective for HE-retrieval of almost 10^{16} data items per month. This is because, initially, the storage cost dominates the overall cost – and so, our solution offers substantial savings; but, gradually, the runtime cost becomes the dominating factor (as storage is constant in our experiment, whereas the runtime is linearly increasing in the number of retrieved data items), leading to a linear cost overhead in our solution.

5.4 Optimization: Parallelization & Streaming

The retrieval runtime in our solution can be significantly and easily optimized by parallelization. In our current implementation, all operations are performed sequentially. They can

Table 4: Microbenchmarks

Number of data items	Runtime (seconds)										HTTP Communication		
	Data Producer			Computing Server			Auxiliary Server						
	Share	AES encrypt	Upload	Process HTTP response	HMAC share derivation	Recon-struct	Down-load	AES decrypt	HE encrypt	Prepare HTTP response	Size (MB)	Runtime (seconds)	
												single cloud	multi cloud
2	0.00001	0.00001	0.02	0.01	0.00001	0.004	0.01	0.000003	0.01	0.01	0.7	0.001	0.1
4,096	0.003	0.00005	0.06	0.01	0.003	0.004	0.02	0.00001	0.01	0.009	0.7	0.001	0.1
16,384	0.01	0.0002	0.06	0.03	0.01	0.02	0.02	0.00004	0.06	0.04	2.7	0.001	0.09
98,304	0.08	0.001	0.1	0.2	0.07	0.09	0.03	0.0002	0.3	0.2	16.0	0.010	0.13
507,904	0.4	0.005	0.3	1.0	0.5	0.5	0.08	0.001	1.7	1.2	83.0	0.06	0.14
2,031,616	1.5	0.02	0.4	3.9	1.5	1.9	0.3	0.01	6.7	4.8	331.0	0.25	0.48

be fully parallelized so that each thread handles a single HE batch (for encryption) or HE ciphertext (for serialization, deserialization, reconstruction), that make up together about 78% of the retrieval runtime. Namely, using AWS EC2 instances with 96 vCPUs, we can reduce 78% of the current runtime by a factor of 96.

Further optimization can be achieved by handling the data sent from Auxiliary Server to Computing Server as a stream. In the current implementation, Computing Server waits for the HTTP response to be returned from Auxiliary Server before it starts to process it. In contrast, with streaming implementation, the Computing Server will process the HE ciphertexts as it gets them, while, in parallel, Auxiliary Server still generates the remaining ones. This will yield an overall running time that is the maximum of the Computing and Auxiliary servers runtime rather then their sum, which amounts to roughly 40% reduction in the total runtime.

6 Conclusions

We presented a compact storage with HE-retrieval solution that eliminates the high storage overhead associated with storing HE-ciphertexts. Our solution attains: compact storage, equal to storing AES ciphertexts; fast runtime, nearly as fast as directly storing and retrieving HE ciphertexts; dynamic control, at the time of retrieval, of the data items to be retrieved and the HE parameters and batching profile; compatibility to common industry's architecture tools and best-practices; and rigorous security analysis. As a central tool we introduce the first perfect secret sharing scheme with efficient homomorphic reconstruction over the reals. We implemented our solution into a system running on AWS EC2 instances and S3 storage. Further major performance speedup is available via parallelization and streaming. This gives a viable solution for use-cases requiring homomorphic computation on sensitive data in long-term storage.

7 Acknowledgment

The authors thank Yaron Sheffer for sharing his knowledge on industry best practices and for helpful discussions. Many thanks also to the anonymous reviewers for many insightful comments and suggestions. This work was supported in part by the Israel Science Foundation grant 3380/19 and the Israel National Cyber Directorate.

References

[1] Aws pricing. https://aws.amazon.com/ec2/pricing/on-demand/. Accessed: 2022-12-18.

[2] Geekbench Browser. https://browser.geekbench.com/. Accessed: 2022-12-14.

[3] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[4] Reform of EU data protection rules, May 2018.

[5] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (CSUR)*, 51(4):1–35, 2018.

[6] Adi Akavia, Max Leibovich, Yehezkel S. Resheff, Roey Ron, Moni Shahar, and Margarita Vald. Privacy-preserving decision trees training and prediction. *ACM Trans. Priv. Secur.*, 25(3), may 2022.

[7] Adi Akavia, Hayim Shaul, Mor Weiss, and Zohar Yakhini. Linear-regression on packed encrypted data in the two-server model. In *WAHC*, 2019.

[8] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *EUROCRYPT*, 2015.

[9] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *NDSS*, 2013.

[10] Varunya Attasena and Nouria Harbi. Secret sharing for cloud data security: a survey. *The VLDB Journal*, 26:657–681, 2017.

[11] George Robert Blakley. Safeguarding cryptographic keys. In *Managing Requirements Knowledge, International Workshop on*, pages 313–313, 1979.

[12] GR Blakley and Laif Swanson. Security proofs for information protection systems. In *1981 IEEE Symposium on Security and Privacy*, pages 75–75. IEEE, 1981.

[13] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In *WAHC*, 2019.

[14] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In *ACM CF*, 2019.

[15] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A **fr**amework for fast privacy-preserving computations. 2008.

[16] Zvika Brakerski, Nico Döttling, Sanjam Garg, and Giulio Malavolta. Leveraging linear decryption: Rate-1 fully-homomorphic encryption and time-lock puzzles. In *TCC*, 2019.

[17] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, pages 309–325, 2012.

[18] Preston Bukaty. *The California Consumer Privacy Act (CCPA): An Implementation Guide*. IT Governance Publishing, 2019.

[19] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. In *FSE*, 2016.

[20] Octavian Catrina. Efficient secure floating-point arithmetic using shamir secret sharing. In *ICETE (2)*, pages 49–60, 2019.

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

[22] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In *FC*, 2010.

[23] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*, 2017.

[24] Jung Hee Cheon, Duhyeong Kim, Yongdai Kim, and Yongsoo Song. Ensemble method for privacy-preserving logistic regression based on homomorphic encryption. *IEEE Access*, 6:46938–46948, 2018.

[25] Jihoon Cho, Jincheol Ha, Seongkwang Kim, ByeongHak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. Transciphering framework for approximate homomorphic encryption. In *ASIACRYPT*, 2021.

[26] Benny Chor and Eyal Kushilevitz. Secret sharing over infinite domains. *Journal of Cryptology*, 6(2):87–95, 1993.

[27] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD$\mathbb{Z}_{2^k}$: Efficient mpc mod 2^k for dishonest majority. In *CRYPTO*. Springer, 2018.

[28] Alexander Dibert and László Csirmaz. Infinite secret sharing–examples. *Journal of Mathematical Cryptology*, 8(2):141–168, 2014.

[29] Vassil Dimitrov, Liisi Kerik, Toomas Krips, Jaak Randmets, and Jan Willemson. Alternative implementations of secure real numbers. In *ACM CCS*, 2016.

[30] Yarkin Doröz, Yin Hu, and B. Sunar. Homomorphic aes evaluation using ntru. *IACR Cryptol. ePrint Arch.*, 2014:39, 2014.

[31] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

[32] Craig Gentry and Shai Halevi. Compressible FHE with applications to PIR. In *TCC*, 2019.

[33] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, 2012.

[34] Irene Giacomelli, Somesh Jha, Marc Joye, C. David Page, and Kyonghwan Yoon. Privacy-preserving ridge regression with only linearly-homomorphic encryption. In *ACNS*, 2018.

[35] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, 2016.

[36] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 307–328. 2019.

[37] Jincheol Ha, Seongkwang Kim, Wonseok Choi, Jooyoung Lee, Dukjae Moon, Hyojin Yoon, and Jihoon Cho. Masta: An he-friendly cipher using modular arithmetic. *IEEE Access*, 8:194741–194751, 2020.

[38] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N Wright. Privacy-preserving machine learning as a service. *Proc. Priv. Enhancing Technol.*, 2018(3):123–142, 2018.

[39] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *ACM CCS*, 2018.

[40] Charanjit Singh Jutla and Nathan Manohar. Sine series approximation of the mod function for bootstrapping of approximate he. Cryptology ePrint Archive, Report 2021/572, 2021. https://ia.cr/2021/572.

[41] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020.

[42] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC medical genomics*, 11(4):23–31, 2018.

[43] Miran Kim, Yongsoo Song, Shuang Wang, Yuhou Xia, Xiaoqian Jiang, et al. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR medical informatics*, 6(2):e8805, 2018.

[44] Hugo Krawczyk. Secret sharing made short. In *Annual international cryptology conference*, pages 136–146. Springer, 1993.

[45] Toomas Krips and Jan Willemson. Hybrid model of fixed and floating point numbers in secure multiparty computations. In *International Conference on Information Security*, pages 179–197. Springer, 2014.

[46] Rakesh Kumar and Rinkaj Goyal. On cloud security requirements, threats, vulnerabilities and countermeasures: A survey. *Computer Science Review*, 33:1–48, 2019.

[47] Yongwoo Lee, Joon-Woo Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and HyungChul Kang. High-precision bootstrapping for approximate homomorphic encryption by error variance minimization. In *CRYPTO*, 2022.

[48] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In *ACM CCS*, 2017.

[49] Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. Pegasus: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. pages 1057–1073, 2021.

[50] Pierrick Méaux, Claude Carlet, Anthony Journault, and François-Xavier Standaert. Improved filter permutators for efficient FHE: better instances and implementations. In *INDOCRYPT*, 2019.

[51] Pierrick Méaux, Anthony Journault, François-Xavier Standaert, and Claude Carlet. Towards stream ciphers for efficient FHE with low-noise ciphertexts. In *EUROCRYPT*, 2016.

[52] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *ACM CCSW*, 2011.

[53] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Eurocrypt*, 1999.

[54] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. Heax: An architecture for computing on encrypted data. In *ASPLOS*, 2020.

[55] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[56] Amartya Sanyal, Matt Kusner, Adria Gascon, and Varun Kanade. Tapas: Tricks to accelerate (encrypted) prediction as a service. In *International Conference on Machine Learning*, pages 4490–4499. PMLR, 2018.

[57] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, November 2020. Microsoft Research, Redmond, WA.

[58] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[59] Katrine Tjell and Rafael Wisniewski. Privacy in distributed computations based on real number secret sharing. *arXiv preprint arXiv:2107.00911*, 2021.

[60] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*.

[61] Lijing Zhou, Ziyu Wang, Xiao Zhang, and Yu Yu. Head: an fhe-based outsourced computation protocol with compact storage and efficient computation. Cryptology ePrint Archive, Report 2022/238, 2022.

A Proof of Theorem 4.1

Proof of Theorem 4.1. Let $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ be a $\mathcal{C}$-homomorphic CPA-secure public key encryption scheme, let $\mathcal{S} = (\mathsf{Shr}, \mathsf{Rec})$ be $\mathcal{E}$-friendly secret sharing scheme with random 1st share for domain A, and let $\lambda \in \mathbb{N}$ be a security

parameter and f_k be a function sampled uniformly at random from a pseudorandom function family $\mathcal{F} = \{f_k \colon \{0,1\}^* \to \mathcal{S}_1\}_{k\in\{0,1\}^\lambda, \lambda\in\mathbb{N}}$ with $|k| = \lambda$. Let $\pi = (\mathsf{store}, \mathsf{retrieve})$ be our protocol from Figure 4 when instantiated with $\mathcal{E}$ and $\mathcal{S}$ for the HE and secret sharing schemes, and with $\mathcal{F}_{\text{storage}}$ and $\mathcal{F}_{\text{keys}}$ for the storage and key management functionalities.

The ppt complexity of dataProd, compSrv, and auxService follows from all components ($(\mathsf{Shr}, \mathsf{Rec})$, f_k, Enc, and Eval) being ppt algorithms.

The correctness of π stems from the correctness and $\mathcal{E}$-friendliness of the secret sharing, and the $\mathcal{C}$-homomorphism of $\mathcal{E}$. More formally, fix some $x \in \mathsf{A}$, $index \in \{0,1\}^*$ and $\lambda \in \mathbb{N}$, and let $s_1 \leftarrow f_k(index)$ and $s_2 \leftarrow \mathsf{Shr}_{s_1}(x)$. The $\mathcal{E}$-friendliness together with $\mathcal{C}$-homomorphism of $\mathcal{E}$ guarantees that for every (pk, sk) in the range of $\mathsf{Gen}(1^\lambda)$ it holds that

$$\Pr\left[\mathsf{Dec}_{sk}\left(\mathsf{Eval}_{pk}\left(\mathsf{Rec}_{s_1}, \mathsf{Enc}_{pk}(s_2)\right)\right) \neq \mathsf{Rec}_{s_1}(s_2)\right] \leq \mathsf{neg}(\lambda) \tag{1}$$

In addition, the correctness of $\mathcal{S} = (\mathsf{Shr}, \mathsf{Rec})$ and the range of $f_k \colon \{0,1\}^* \to \mathcal{S}_1$ guarantees correctness of the reconstructed value when sampling s_1 is done with f_k, and therefore

$$\mathsf{Rec}_{s_1}(s_2) = \mathsf{Rec}(s_1, s_2) = x \tag{2}$$

Combining together Equations 1-2 we obtain

$$\Pr\left[\mathsf{Dec}_{sk}\left(\mathsf{Eval}_{pk}\left(\mathsf{Rec}_{s_1}, \mathsf{Enc}_{pk}(s_2)\right)\right) \neq x\right] \leq \mathsf{neg}(\lambda) \tag{3}$$

Moreover, the construction of π defines the output of compSrv to be $\mathsf{out}^{\pi}_{\mathsf{compSrv}}(x;\, index, \lambda) = \mathsf{Eval}_{pk}\left(\mathsf{Rec}_{s_1}, \mathsf{Enc}_{pk}(s_2)\right)$ and hence implies together with Equation 3 the correctness of π, i.e., $\Pr\left[\mathsf{Dec}_{sk}\left(\mathsf{out}^{\pi}_{\mathsf{compSrv}}(x;\, index, \lambda)\right) \neq x\right] \leq \mathsf{neg}(\lambda)$.

The compactness of π follows directly from the 2nd share compactness together with the construction in Figure 4, as for any input x only the second share s_2 is being stored and $|s_2| = |x|$.

The privacy proof for any ppt adversary $\mathsf{compSrv}^*$ relies on its view being independent of x (and presumably only depending on the size of x). More formally, consider a construction $\bar{\pi}$ similar to π, where on any $index^*$ received from $\mathsf{compSrv}^*$ instead of Step 2b of retrieve in π the auxiliary service auxService encrypts a random $r \in \mathcal{S}_2$ s.t $|r| = |s_2|$. From the CPA-security of $\mathcal{E}$ we obtain that for any $x \in \mathsf{A}$, $index \in \{0,1\}^*$ and $\lambda \in \mathbb{N}$ the following holds:

$$\mathsf{view}^{\pi}_{\mathsf{compSrv}^*}(x; index, \lambda) \approx_c \mathsf{view}^{\bar{\pi}}_{\mathsf{compSrv}^*}(x; index, \lambda) \tag{4}$$

In addition, the first share as derived in an execution of $\bar{\pi}$ depends only on $f_k(\cdot)$ and is independent of x. Therefore, for any $\lambda \in \mathbb{N}$, every ppt distinguisher $\mathcal{D}$ that chooses $x, x' \in \mathsf{A}$ s.t. $|x| = |x'|$ and any $index$ the following holds:

$$\left|\Pr[\mathcal{D}(\mathsf{view}^{\bar{\pi}}_{\mathsf{compSrv}^*}(x; index, \lambda)) = 1] - \Pr[\mathcal{D}(\mathsf{view}^{\bar{\pi}}_{\mathsf{compSrv}^*}(x'; index, \lambda)) = 1]\right| \leq \mathsf{neg}(\lambda)$$

Combining this with Equation 4 we obtain the desired.

For the privacy proof against any $\mathsf{auxService}^*$, first note that if π is executed on a previously used index then the view of $\mathsf{auxService}^*$ is completely independent of the current execution input data x, as nothing related to it was stored in $\mathcal{F}_{\text{storage}}$. Therefore we consider executions with a unique $index$.

Consider a version of π, denoted by $\bar{\pi}$, where f_k is replaced by sampling $s_1 \leftarrow_R \mathcal{S}_1$ at random. That is, Step 1 of store in Figure 4 is replaced with $s_1 \leftarrow_R \mathcal{S}_1$ and $(\mathsf{Upload}, index, s_1, \mathsf{compSrv})$ to $\mathcal{F}_{\text{storage}}$ and Step 1 in retrieve is replaced with sending $(\mathsf{Download}, index)$ to $\mathcal{F}_{\text{storage}}$ to get s_1. Since $\bar{\pi}$ is executed on a unique $index$ it holds that for any x and $index$ the distribution of shares (s_1, s_2) produced by Shr is identical to the distribution produced by sampling s_1 with a uniformly, randomly sampled function R from the set of all functions $\{R : \{0,1\}^* \to \mathcal{S}_1\}$ and having $s_1 \leftarrow R(index)$. Moreover, the pseudorandomness property guarantees that for every x, the shares (s_1, s_2) produced by f_k are distributed computationally close to shares produced by R. Therefore, for any $\lambda \in \mathbb{N}$, $x \in \mathsf{A}$ and any $index$ the following views are indistinguishable,

$$\mathsf{view}^{\pi}_{\mathsf{auxService}^*}(x; index, \lambda) \approx_c \mathsf{view}^{\bar{\pi}}_{\mathsf{auxService}^*}(x; index, \lambda) \tag{5}$$

Next fix some value $y \in \mathsf{A}$ and consider a version of $\bar{\pi}$, denoted by $\bar{\pi}^y$, with a modified storage functionality $\mathcal{F}^y_{\text{storage}}$ that behaves as $\mathcal{F}_{\text{storage}}$ besides that on any download request from $\mathsf{auxService}^*$ with respect to any $index$ that is stored it computes $s_1 \leftarrow R(index)$ and $s_2 \leftarrow \mathsf{Shr}_{s_1}(y)$ and returns s_2. By the perfect security of $(\mathsf{Shr}, \mathsf{Rec})$ and a hybrid argument it is guaranteed that for any $\lambda \in \mathbb{N}$, $x \in \mathsf{A}$ and every $index$

$$\mathsf{view}^{\bar{\pi}}_{\mathsf{auxService}^*}(x; index, \lambda) \equiv \mathsf{view}^{\bar{\pi}^y}_{\mathsf{auxService}^*}(x; index, \lambda) \tag{6}$$

Note that the view of any $\mathsf{auxService}^*$ in $\bar{\pi}^y$ is completely independent of the input x since in $\mathcal{F}^y_{\text{storage}}$ the reply of s_2 is derived from a fixed and independent value y. Therefore, we obtain that for any $\lambda \in \mathbb{N}$, every $x, x' \in \mathsf{A}$ s.t $|x| = |x'|$ and any $index$ the following holds:

$$\mathsf{view}^{\bar{\pi}^y}_{\mathsf{auxService}^*}(x; index, \lambda) \equiv \mathsf{view}^{\bar{\pi}^y}_{\mathsf{auxService}^*}(x'; index, \lambda)$$

Combining this with Equations 5 and 6 we obtain the desired, i.e., for any $\lambda \in \mathbb{N}$, every ppt distinguisher $\mathcal{D}$ that chooses $x, x' \in \mathsf{A}$ s.t $|x| = |x'|$ and any $index$,

$$\left|\Pr[\mathcal{D}(\mathsf{view}^{\pi}_{\mathsf{auxService}^*}(x; index, \lambda)) = 1] - \Pr[\mathcal{D}(\mathsf{view}^{\pi}_{\mathsf{auxService}^*}(x'; index, \lambda)) = 1]\right| \leq \mathsf{neg}(\lambda)$$

as desired. □