Pantheon: Private Retrieval from Public Key-Value Store

Ishtiyaque Ahmad
University of California Santa Barbara
ishtiyaque@ucsb.edu

Divyakant Agrawal
University of California Santa Barbara
agrawal@cs.ucsb.edu

Amr El Abbadi
University of California Santa Barbara
amr@cs.ucsb.edu

Trinabh Gupta
University of California Santa Barbara
trinabh@ucsb.edu

ABSTRACT

Consider a cloud server that owns a key-value store and provides a private query service to its clients. Preserving client privacy in this setting is difficult because the key-value store is *public*, and a client cannot encrypt or modify it. Therefore, privacy in this context implies hiding the access pattern of a client. Pantheon is a system that cryptographically allows a client to retrieve the value corresponding to a key from a *public* key-value store without allowing the server or any adversary to know any information about the key or value accessed. Pantheon devises a single-round retrieval protocol which reduces server-side latency by refining its cryptographic machinery and massively parallelizing the query execution workload. Using these novel techniques, Pantheon achieves a 93× improvement for server-side latency over a state-of-the-art solution.

PVLDB Reference Format:
Ishtiyaque Ahmad, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Pantheon: Private Retrieval from Public Key-Value Store. PVLDB, 16(4): 643 - 656, 2022.
doi:10.14778/3574245.3574251

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ishtiyaque/Pantheon.

1 INTRODUCTION

Due to the widespread use of cloud applications, searching for data from a cloud server has become ubiquitous. However, accessing data stored in a cloud server comes with severe privacy concerns owing to numerous attacks and data breaches [8, 15, 37, 67]. A long line of work [11–14, 19, 28, 36, 41, 48, 54–56, 58, 63, 66, 68, 70] (§6) addresses this privacy concern for *private data* where a client owns the data and outsources it in encrypted form to a cloud server. The server then executes client queries on the encrypted data to ensure privacy. However, none of these approaches provide privacy for querying over *public data*, where the data is owned and managed by a cloud server, and the server provides query services to many clients. A practical use case is a breached password database service, where a client may query whether her password has been breached and other relevant information. However, the client may be unwilling to disclose which password she is querying about to the service provider or any network eavesdropper. Similarly, when customers query information about a particular ticker symbol from a stockbroker, or information about a certain disease from a medical repository, they may want to hide the query keywords to preserve their financial and medical privacy. A common feature in all these use-cases is that the server holds a key-value store and clients query about keys. Evidently, a client cannot encrypt the data, i.e., the key-value store in this setting. Therefore, information on which key a client performs queries can compromise privacy [47, 69]. Unfortunately, preserving client privacy in this application domain of *public data* has received very little attention in the literature. This paper addresses this general research problem where a cloud server owns and manages a key-value store, and the clients want to perform queries without sacrificing privacy.

The problem of private retrieval from public data is closely associated with Private Information Retrieval (PIR) [22, 23, 42]. At a high level, PIR allows a client to retrieve an element from an untrusted server without letting the server know which element the client retrieved. However, PIR requires the server to consider the data as an array of elements and the client to know the array-index of the desired element (for example, the client wants to retrieve the element at index 13 from an array of 100 elements). This requirement is a limiting factor in many practical use cases, especially for key-value stores, where the client may be interested in a particular key, but does not know the exact arrangement of the data at the server. An extension of PIR, known as keyword-PIR [21], bypasses this restriction by using multiple rounds of PIR. It allows a client to retrieve the value corresponding to a key from an untrusted server obliviously. Nevertheless, keyword-PIR requires the client to know the total number of keys (n) in the key-value store and perform $\lceil \log_2(n+1) \rceil + 1$ sequential PIR interactions with the server. As a result, it suffers from three significant limitations. First, the number of round-trips increases with the number of keys and thus creates performance and scalability bottlenecks. For example, the keyword-PIR protocol requires 21 round-trips to retrieve a value from a key-value store containing 1M tuples. Second, the client must know the number of tuples n in the key-value store before constructing a query, creating performance overhead for a dynamic key-value store. Finally, the keyword-PIR protocol involves $O(\log n)$ round-trips between the server and client, where each round requires processing the key-value store. Therefore, the server must preserve its state across the rounds of a query to guarantee consistency. Naturally, one would prefer to retrieve the value corresponding to a key in a single-round to guarantee consistency

Proceedings of the VLDB Endowment, Vol. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574251

and atomicity. In this paper, we focus on single-round solutions to the private key-value retrieval problem.

A general approach for constructing a single round solution is to use Fully Homomorphic Encryption (FHE) [17, 31]. In this approach, a client constructs a query q that cryptographically hides the desired key k using FHE and the client sends q to the server. The server obliviously checks equality between k (hidden inside q) and each key in the key-value store using a homomorphic equality operator to determine an encrypted representation of the index of k in the key-value store. Then, the server uses this hidden index information and Private Information Retrieval (PIR) [22, 23, 42] to obtain an encrypted form of the desired value as the response. The client receives the server's response and converts it into the plaintext form of the value.

A number of prior works [5, 6, 27, 32, 46] adopted this approach with different techniques for equality check and PIR. A recent work named Constant-weight Keyword PIR (CKP) [46] is so far the best known instantiation of this FHE-based single-round approach. It proposes a new homomorphic equality operator to check equality between the query key and the keys in the key-value store. Then, it uses the SealPIR [9] technique to retrieve the value using Private Information Retrieval. Even though this work offers a single-round solution to the private key-value retrieval problem, it has major limitations in terms of performance and scalability. First, their proposed equality operator, though better than prior works, involves expensive homomorphic operations. At a high level, the constant-weight equality operator obliviously evaluates a boolean circuit, and therefore requires each bit of the operand to be encrypted separately into a different FHE ciphertext. The computation for the equality operator comprises expensive homomorphic multiplication operations, and the number of homomorphic multiplications required is a multiple of the number of tuples in the key-value store. Second, the output of the equality operator also involves one ciphertext for each result, and therefore leads to a costly PIR technique. For example, using an AWS instance containing 48 vCPU as the server, the latency for retrieving a value privately from a key-value store containing 64K tuples is 107.8 seconds (§5.2), out of which the equality checking takes 80.5 seconds and the PIR step takes 24.9 seconds.

This paper addresses the *performance* and *scalability* issues of privately querying over public data. We present Pantheon, a system that provides a single-round solution to the private key-value retrieval problem and scales to millions of tuples while keeping the server-side latency reasonably low. Pantheon achieves this performance and scalability with contributions in two directions - it refines the cryptographic machinery of the single-round protocol, and applies system level optimizations to support scalability. The primary cryptographic contribution of Pantheon is to present a new homomorphic equality operator using Fermat's little theorem [57]. The key advantage of Pantheon's equality operator is that it is a number theoretic technique, thus the computation is performed in an integer space rather than requiring the evaluation of a bitwise boolean circuit. In addition, Pantheon takes advantage of the SIMD batching property of FHE to pack multiple operands in the same ciphertext and thus amortizing the cost. Due to these two techniques, Pantheon's equality operator reduces the number of ciphertext multiplications compared to CKP by three orders of magnitude (§5.1), resulting in significantly lower computation. However, scaling Pantheon to support practical large key-value stores still remains a monumental challenge. This challenge is due to a fundamental lower bound on the server-side computation. More specifically, while serving a client's query, the Pantheon server must process the entire key-value store; otherwise, it will learn information about the client's query. Therefore, scaling the system leads to high computational overhead on the Pantheon server. Pantheon addresses this systems level challenge by carefully distributing its workload over a cluster of machines and massively parallelizing the computation in each machine. Note that, Fermat's little theorem has been known for centuries, but Pantheon makes the first use of it to provide an end to end solution in a practically scalable manner.

We have implemented (§4) and evaluated (§5) a prototype of Pantheon. Our implementation includes parallelizing part of the state-of-the-art homomorphic encryption library Microsoft SEAL [60], thus enabling the Pantheon server to distribute its computation over a cluster of machines and multiple cores in a single machine. When the Pantheon server is deployed on a single AWS instance containing 48 vCPU, the latency for performing a GET query from a key-value store containing 64K tuples is 1.15 seconds (§5.2), 93× better than the to-date best system built on Constant-weight Keyword PIR [46]. We also deploy the Pantheon server over a cluster of AWS instances, and the latency for a private GET query from a key-value store containing 2M tuples is 0.99 seconds (§5.3). Indeed, the latency is substantially higher than a non-private system. We deem this increase in latency as the cost of privacy. However, to our knowledge, Pantheon is the first system to support private retrieval from a million-scale public key-value store with sub-second latency. Moreover, Pantheon is vertically (§4.1) and horizontally (§4.2) scalable. One can reduce the latency by adding computational resources and leveraging Pantheon's parallelization capability.

2 PROBLEM OVERVIEW

Pantheon addresses a setting where an untrusted server owns a key-value store and provides a query service to its clients. The server performs the write operations– PUT, UPDATE, and DELETE. The clients issue GET queries for any key. Since write operations are performed by the server, client-side privacy only concerns read operations, i.e., GET queries. The goal is to hide the access pattern, i.e., which key or value a client is interested in, from the server or any adversary. In this section, we formalize the problem, state solution goals, and provide an overview of possible solution approaches.

2.1 Problem formulation

A server has a set S of key-value pairs $\{(k_1, v_1), (k_2, v_2), ..., (k_n, v_n)\}$ where each key in $\{k_1, k_2, ..., k_n\}$ is unique, i.e., the keys are primary keys. The server stores the content of S in plaintext and can insert, update, or delete key-value tuples from S. A client holds a key k and wants to know the corresponding value v if $k \in \{k_1, k_2, ..., k_n\}$ such that $(k, v) \in S$, or an empty value otherwise. During this retrieval, the server or any other network eavesdropper must not be able to know anything about k and also not distinguish between returning an empty value and some other value in $\{v_1, v_2, ..., v_n\}$. The server should be able to serve queries from multiple independent clients who may not trust each other.

2.2 Threat Model

Pantheon assumes a *passive-adversary* threat model to guarantee query privacy and result integrity. The adversary may see the content of the key-value store, monitor and store all the queries and responses associated with the clients, and perform any analysis on them. It may monitor and analyze any operation performed by the server and perform side-channel attacks on the server. The adversary may also monitor, log, and analyze any network traffic. We assume the adversary is not actively malicious, i.e., it does not corrupt the key-value store, tamper with any server-side computation, or modify network traffic. Any such adversary may generate incorrect response and thus violate integrity. However, Pantheon must guarantee that no information about the client's query is leaked even in the presence of an *active-adversary* that can arbitrarily modify any data or computation.

We assume the adversary cannot compromise or perform side-channel attacks on the client, because in that case knowing the client query key becomes trivial. We also consider the adversary cannot break standard cryptographic assumptions, such as the semantic security of an encryption scheme.

2.3 Goals

2.3.1 ***Query privacy.*** Pantheon must guarantee *query privacy* to its clients. An adversary must not be able to learn any information about a client's query key k. It also implies the adversary must not learn any information about the value retrieved.

2.3.2 ***Consistency.*** Any response received by a client should be from a stable version of the key-value store that incorporates a write in its entirety or not at all, i.e., it should not involve *dirty* data. We materialize this goal by making Pantheon a single-round protocol.

2.3.3 ***Performance and scalability.*** The primary performance metric for a system like Pantheon is its server-side latency for serving a Get query. In addition, the system should be scalable with three parameters related to the size of the key-value store, namely, the size of each key, the size of each value, and the number of tuples in the key-value store (n). For instance, Pantheon should support at least 256-bit keys, so that any arbitrary size key-object can be mapped to a collision-resistant hash digest of the object using commonly known hash functions such as SHA-256 [52]. Further, Pantheon should support arbitrarily large values, because hashing a value does not serve the purpose in most cases. The Pantheon server should also be able to serve a Get query with a reasonable latency, say a few seconds, when the key-value store contains several millions of tuples.

Pantheon's client-side protocol for performing a Get query should be independent of the total number of tuples (n) in the key-value store. Otherwise, it creates a substantial overhead for dynamic key-value stores where the value of n changes frequently.

2.4 Possible solution approaches

Before going into the details of Pantheon's architecture, we discuss some possible solution approaches to develop the intuition. We also discuss the limitations of these approaches to rationalize the need for a system like Pantheon.

2.4.1 ***Strawman 1: Download the entire key-value store.*** The client may download the entire key-value store S and search for (k, v) locally. This approach may be suitable if the size of S is small, but not otherwise.

2.4.2 ***Strawman 2: Client downloads the key-set, then performs Private Information Retrieval.*** Usually, the size of a key is smaller than the value. In that case, the client can first download the entire set of keys, and find the index of k in the key-set locally. Then, the client may privately retrieve the value at that index from the server using Private Information Retrieval (PIR) [22, 23, 42]. There are two major problems with this approach. First, it is a multi-round protocol and therefore does not guarantee consistency (§2.3.2). Second, the client-side download increases with the size of the key-set, making the approach difficult to scale. For example, for a key-value store containing 2M tuples where each key is 256-bits, a client has to download 64MiB of data just for the first round of the protocol.

2.4.3 ***Keyword-PIR.*** Chor et al. [21] propose a protocol to retrieve the value corresponding to a key from an untrusted server using multiple sequential rounds of PIR. Their protocol, known as keyword-PIR, proceeds in two phases. In the first phase, the client performs $\lceil \log_2(n+1) \rceil$ round-trip PIR interactions with the server to find out the index of the desired key, where n is the total number of keys. Then, in the second phase, the client uses this index to retrieve the value using another round of PIR. This protocol may be preferable to the previous strawman approach if the number of keys is too large to download in its entirety. However, the number of rounds in this protocol increases with the number of keys, making it a scalability bottleneck. In addition, the protocol cannot guarantee the desirable consistency property (§2.3.2). Furthermore, the keyword-PIR protocol requires the client to know the total number of tuples (n) in the key-value store to initiate a query, which either adds one more round to the protocol for fetching the latest value of n or makes the server broadcast n after every write operation.

2.4.4 ***Homomorphic encryption.*** The primary challenge to building a single-round solution to the private key-value retrieval problem is obliviously checking equality between the client query key and the keys in the key-value store. After that, the result of the equality check can be utilized to retrieve the value using Private Information Retrieval (PIR) [22, 23, 42]. Conceptually, it is possible to check the equality obliviously using Fully Homomorphic Encryption (FHE) [17, 31]. We discuss the multiple approaches in the literature using homomorphic encryption as follows.

Evaluating fully homomorphic boolean function. In theory, FHE can evaluate any boolean function over an encrypted data and generate an encrypted output of the function. There are a number of approaches [5, 6, 27, 32, 46] that express the equality operator as a boolean function and evaluates that function homomorphically. However, these techniques are limited in terms of performance and scalability mainly because the boolean function operates over each bit of the operands individually and therefore involve prohibitively large number of homomorphic operations.

The state-of-the-art single-round solution to the private key-value retrieval problem is a protocol named Constant-weight Keyword PIR (CKP) [46]. This protocol proposes a new boolean operator named constant-weight equality operator for homomorphic

equality check. This equality operator requires all the keys to have the same number of 1's in their binary representations, i.e., the hamming weight of the keys need to be the same (hence, constant-weight). Accordingly, CKP applies a transformation to map each key to a constant-weight binary string of larger size. For instance, when the key-size is 32-bit, CKP converts each key to a 2955-bit string having hamming-weight (h) as 3. As a result, the protocol requires an expensive initial step where the client's query is expanded to a large number of ciphertexts (2955 ciphertexts for 32-bit key-size), each encrypting one bit (§5.1). Afterwards, the equality check requires $n(h-1)$ homomorphic multiplication operations, where n is the number of tuples in the key-value store. This huge number of operations make the performance impractical, since homomorphic operations are generally computationally expensive. For example, using an AWS EC2 instance of type `c5.12xlarge` (48 vCPU, 96 GiB of RAM) as the server and populating the key-value store with 64K tuples where each key is 32 bits and each value is 256 bytes long, the latency for privately retrieving a value is 107.8 seconds (§5.2). Another major limitation with CKP is, it does not scale with the size of the key or the number of tuples in the key-value store. For instance, the current implementation of the work [24] does not support keys larger than 60 bits and is not horizontally scalable. Lastly, the Constant-weight Keyword protocol requires the client to know the total number of tuples (n) in the key-value store for constructing a query. As a result, it introduces performance overhead for dynamic key-value stores.

Using number theoretic technique. Another way of performing a homomorphic equality check is to use technique from number theory such as Fermat's little theorem [57]. This approach considers keys as integers and does not require bitwise encryption of the key. However, this technique involves homomorphic exponentiation, which is also computationally expensive. Therefore, a straightforward implementation using this approach yields impractical performance as well. An example developed by HElib [40] is the best available solution that uses Fermat's little theorem for equality check. Nevertheless, this example implementation takes more than 10 seconds in a single machine to perform a query over a key-value store containing (country name, capital) tuples for 47 European countries, which is worse than the strawman solution of downloading the entire key-value store (§2.4.1). Overcoming the performance bottlenecks and scaling this technique to support millions of key-value tuples remains an open problem. Pantheon takes inspiration from this approach and uses Fermat's little theorem for oblivious equality check (§3.4.2). However, Pantheon drastically improves the performance to support queries over a key-value store containing millions of tuples with two key contributions. First, it carefully refines the cryptographic components to reduce the number of expensive homomorphic multiplications by three orders of magnitudes (§5.1). Then, Pantheon parallelizes its workload to make the system both vertically and horizontally scalable (§5.3).

3 PANTHEON DESIGN

In this section, we discuss Pantheon's design in detail. First, we discuss the architecture of Pantheon's protocol (§3.1) and the cryptographic constructs Pantheon relies on (§3.2). Next, we explain how a new client registers itself with the Pantheon server (§3.3).

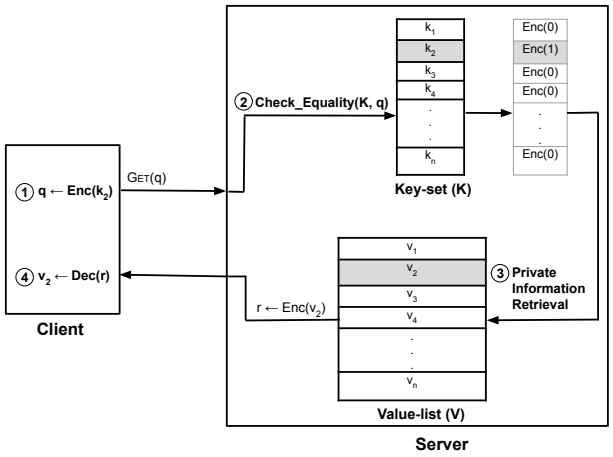


Figure 1: **High-level architecture of Pantheon.**

After that, we elaborate on the steps for privately querying with a key (§3.4). Then, we devise a query compression technique (§3.5) that optimizes Pantheon's network overhead. Finally, we provide a security analysis of Pantheon's protocol (§3.6).

3.1 Basic Architecture

Figure 1 shows Pantheon's high-level architecture. Pantheon consists of an untrusted server and its clients. The server holds a key-value store $\{(k_1, v_1), (k_2, v_2), (k_3, v_3), ..., (k_n, v_n)\}$ denoted by S. For operational flexibility, the server stores the keys $\{k_1, k_2, k_3, ..., k_n\}$ into an array K and the values $\{v_1, v_2, v_3, ..., v_n\}$ into an array V such that for any $(k_i, v_i) \in S$, $K[i] = k_i$ and $V[i] = v_i$. Pantheon's server-side operations to serve a query require all the keys in K to be of the same size and all the values in V to be of the same size. The server applies appropriate padding to the keys and the values, if required, to satisfy this condition.

A Pantheon client can retrieve the value corresponding to a key of interest k from the server in a single round-trip between the client and the server. It takes place in four steps (as shown in Figure 1). In step 1, client encodes the query key k into q using Pantheon's encoding procedure that ensures q does not reveal any information about k to an adversary. Then, the client calls the Pantheon server's GET API with q. Steps 2 and 3 take place at the server end. In step 2, the Pantheon server runs an oblivious equality check with each key in K and the client's encoded query q. For each key in K, the equality check outputs an encryption of 1 if it equals the client's query key k or an encryption of 0 otherwise. Step 3 uses the output of step 2 to perform Private Information Retrieval (PIR) on the value-list V. PIR outputs an encryption of the desired value if $k \in K$, or the encryption of an empty value otherwise. This encrypted value is then sent back to the client. In step 4, the client uses Pantheon's decode procedure to convert the encrypted value to its plaintext form.

Pantheon relies on homomorphic encryption to provide its privacy guarantees. More specifically, we use the BFV scheme [16, 29] of homomorphic encryption because it is standardized [7], resilient to quantum attacks, and has an actively maintained open-source implementation [60]. In this section, we first give a brief introduction to the BFV homomorphic encryption scheme, and then a detailed discussion of Pantheon's functionalities.

3.2 Basics of BFV homomorphic encryption

Pantheon uses the more efficient vectorized variant of BFV that allows operating over a vector of data simultaneously and takes advantage of the Single Instruction Multiple Data (SIMD) programming model. In this variant of BFV, a plaintext is a vector of dimension N, where N can be any value in $\{2^{10}, 2^{11}, ..., 2^{15}\}$ [7]. Each element in the plaintext vector is from a set of integers modulo a prime p, i.e., from the set $\{0, 1, ..., p-1\}$. This vector of dimension N is the smallest granularity with which a BFV plaintext can exist. BFV supports an Encrypt procedure that converts the plaintext vector into a ciphertext with the help of the encryptor's secret key. The ciphertext consists of 2 polynomials, each having N coefficients. The ciphertext coefficients are from a set of integers modulo a composite number $p\prime$ such that $p\prime \gg p$. A ciphertext can be decrypted to get the hidden plaintext vector by using BFV's Decrypt method and the secret key that was used for encrypting it in the first place.

BFV encryption supports a number of operations on its ciphertext that eventually modify its underlying plaintext vector. One important point is that all such operations keep the plaintext elements modulo p. Pantheon uses the following homomorphic operations supported by the BFV scheme:

- Add(c_0, c_1) takes two ciphertexts c_0 and c_1 as input which are encryptions of plaintext vectors v_0 and $v_1$respectively, and outputs an encryption of $(v_0 + v_1)$ (component-wise addition).
- Subtract(c_0, c_1) takes two ciphertexts c_0 and c_1 as input which are encryptions of plaintext vectors v_0 and v_1 respectively, and outputs an encryption of $(v_0 - v_1)$ (component-wise subtraction).
- SubtractPlain(c_0, v_1) takes a ciphertext c_0 (encryption of plaintext vector v_0) and a plaintext vector v_1 as input, and outputs an encryption of $(v_0 - v_1)$ (component-wise subtraction).
- Rotate(c, i) takes as input a ciphertext c (encryption of plaintext vector v), an integer $0 < i < N$, and produces a ciphertext c_{out} such that c_{out} is an encryption of v rotated left cyclically by i positions. For example, if $N = 4$ and c encrypts the plaintext (w, x, y, z), then a rotation by $i = 3$ produces a ciphertext that is an encryption of (z, w, x, y).
- Multiply(c_0, c_1) takes two ciphertexts c_0, c_1 as input which are encryptions of plaintext vectors v_0 and v_1 respectively, and outputs an encryption of $(v_0 * v_1)$ (component-wise multiplication).
- MultiplyPlain(c_0, v_1) takes a ciphertext c_0 (encryption of plaintext vector v_0) and a plaintext vector v_1 as input, and outputs an encryption of $(v_0 * v_1)$ (component-wise multiplication).
- Exponentiate(c_0, i) takes a ciphertext c_0 (encryption of plaintext vector v_0), an integer i as input, and outputs an encryption of v_{out} such that, for $0 \leq j < N$, $v_{out}[j] = (v_0[j])^i$. Exponentiate may use Multiply as a subroutine.

3.3 One-time Registration phase

A new client joining Pantheon needs to go through an initial registration phase. During the registration phase, the client and server exchange some one-time information required for serving future queries. First, the server shares three cryptographic parameters: N, p, $p\prime$ (§3.2) and two system parameters: the size of a key and the size of a value with the client. Then, the client constructs its secret key according to the cryptographic parameters. The system parameters are useful for encoding queries (§3.4.1) and decoding responses (§3.4.4). In addition, the client constructs a number of public keys required for performing homomorphic rotation and multiplication operations. Note that, the secret and public keys are cryptographic keys, not to be confused with the ones in the key-value store. The client also encrypts a vector of length N containing all 1's using its secret key. We will refer to this encryption of all 1's as the *one-ciphertext* of the client. The client then shares its public keys and the *one-ciphertext* with the server. The server stores this information corresponding to the client and confirms registration.

3.4 Value Retrieval

The value retrieval protocol in Pantheon takes place in four steps (as shown in Figure 1). We now discuss these steps in detail.

3.4.1 ***Step 1: Encode client query key.*** This section discusses an unoptimized version of Pantheon's query encoding method that takes the client query key k as input and, depending on the size of k, outputs one or more BFV ciphertexts as q. Later, we will present an optimization technique (§3.5) to compress the output query q into a single ciphertext independent of the size of k. Pantheon's query encoding method uses the Encrypt procedure of the BFV encryption scheme(§3.2) to hide k. First, let us consider the simple case where each key is of size t bits, where $t = \lceil \log_2 p \rceil - 1$. Therefore, a key can be represented as an integer in the set $\{0, 1, 2, ..., p-1\}$. To encode the query key k, the client will first construct a plaintext vector of size N with all elements as k and then encrypt this plaintext using the client's secret key. As a toy example, suppose $N = 4$, $p = 17$. Then the length of a key may be at most $t = 4$ bits and so the integer representation of a key will always be smaller than 17. Let us assume that client's query key k is the 4-bit binary string 1100, equivalent to integer value 12. Then the client constructs a plaintext vector $(12, 12, 12, 12)$ and encrypts it using BFV's Encrypt method. The output of the encode method q then consists of this single ciphertext.

Now we extend the query encoding procedure to the case where the size of each key is larger than t bits. Let each key be αt bits, where α is a positive integer. The keys can be padded accordingly if the size is not an integer multiple of t bits. Then, the client can split the query key k into α chunks, each of size t bits. The client then constructs α different ciphertexts with each of these chunks and outputs q as an array of these α ciphertexts. Continuing with the previous example, let $\alpha = 2$ and client's query key is the 8-bit binary string 11001110. The client will then split k into 2 chunks $\{1100, 1110\}$, also represented as $\{12, 14\}$ in integer forms. The client will then encrypt two plaintext vectors $(12, 12, 12, 12)$ and $(14, 14, 14, 14)$, and output the two ciphertexts as q. Note that, this construction is independent of the number of tuples in the key-value store.

3.4.2 ***Step 2: Oblivious equality check.*** After receiving the client's encoded query q, the Pantheon server performs an equality check between q (which is an encryption of client's desired key k) and all the keys in key-set K. Pantheon takes advantage of Fermat's little theorem [57] to perform this equality check obliviously. Fermat's little theorem implies that if p is a prime number and a is a number not divisible by p, then $a^{(p-1)} \equiv 1 \pmod{p}$. For example, if $p = 17$, then for any $0 < a < p$, according to Fermat's

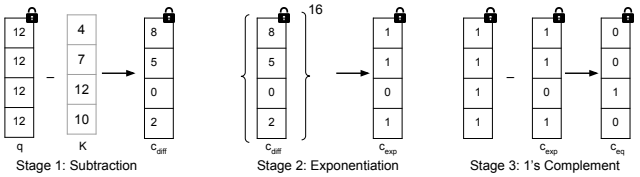


Figure 2: **Three stages of Pantheon's oblivious equality check (assuming N = 4 and p = 17). In stage 1, the server homomorphically subtracts the key-array K form client's query q to obtain c_{diff}. In stage 2, c_{diff} is exponentiated by $(p-1)$ to obtain c_{exp} according to Fermat's little theorem. In stage 3, c_{exp} is subtracted from *one-ciphertext* to obtain the output c_{eq}.**

little theorem, $(a^{16})\%p = 1$. In contrast, if $a = 0$, a^{16} still equals 0; a property to distinguish between a zero and a non-zero value.

For the equality check, let us first consider the simplest case where the key-set K contains N keys, each of size $t = (\lceil \log_2 p \rceil - 1)$ bits. So, each key can be represented as an integer smaller than p, and the client query q consists of a single ciphertext. We will later extend the formulation to support a larger key size and more keys. The goal of the equality operator is to obtain a ciphertext c_{eq} that is an encryption of a plaintext vector v_{eq} of size N such that,

$$v_{eq}[i] = \begin{cases} 1, & \text{if } K[i] == k \\ 0, & \text{otherwise} \end{cases}$$

The oblivious equality check proceeds in three stages as demonstrated in Figure 2.

In stage 1, the server performs $c_{diff} \leftarrow \textsc{SubtractPlain}(q, K)$. As a result, c_{diff} becomes the encryption of a plaintext v_{diff} which contains a zero at index i if $K[i] == k$ and a non-zero value otherwise. In stage 2, Fermat's little theorem is used to distinguish between the encrypted zero and non-zero values. Server computes $c_{exp} \leftarrow \textsc{Exponentiate}(c_{diff}, p-1)$. The resultant c_{exp} will be the encryption of a plaintext vector v_{exp} such that,

$$v_{exp}[i] = \begin{cases} 0, & \text{if } K[i] == k \\ 1, & \text{otherwise} \end{cases}$$

In stage 3, the one's complement of the values in v_{exp} is calculated by homomorphically subtracting c_{exp} from the *one-ciphertext* of the client (§3.3), which is an encryption of a vector containing N 1's. The resultant of the subtraction operation is the desired equality ciphertext c_{eq} as shown in Figure 2.

Now, let us consider the case where each key is larger than $t = (\lceil \log_2 p \rceil - 1)$ bits. Suppose, each key is αt bits long, where α is a positive integer. We can pad each key with the required number of zeros for making the key length an integer multiple of t bits. In this case, the server will first split K into α columns $K_1, K_2, ..., K_\alpha$ such that each K_i contains t bits of each of the keys in K. The client query q also contains α ciphertexts $\{q_1, q_2, ..., q_\alpha\}$ (§3.4.1), each of which encrypts t bits of the client's query key k. Then the server will check equality between each q_i and K_i following the procedure discussed above (3 stages of Figure 2) to get a corresponding resultant ciphertext c_{eq_i}. In order to realize the overall equality between k and the keys in K, a logical *AND* operation among these chunk-wise equality results is required. So, the server computes c_{eq} such that, $c_{eq} = \prod_{i=1}^{\alpha} c_{eq_i}$, where, $\prod$ denotes homomorphic multiplication using Multiply (§3.2).

Now, let us consider the case where the number of keys in K is βN, where β is a positive integer and N is the length of each BFV plaintext (§3.2). If necessary, we can pad K with dummy keys to make the size an integer multiple of N. The resultant of equality operation c_{eq} should now consist of β ciphertexts $\{c_{eq}^1, c_{eq}^2, c_{eq}^3, ..., c_{eq}^\beta\}$. The server horizontally partitions K into β parts $\{K^1, K^2, K^3, ..., K^\beta\}$, each partition containing N keys. Then, for each partition K^j, c_{eq}^j is calculated to contain the equality between q and the partition K^j. Thus, the oblivious equality operator can be extended to support an arbitrary number of keys in K.

3.4.3 ***Step 3: Private Information Retrieval.*** Step 3 uses Private Information Retrieval (PIR) [22, 23, 42] to retrieve the desired value from V. A PIR protocol runs between a PIR server and a PIR client where the PIR server holds an array of n elements and the PIR client is interested in the element at index i. PIR allows the PIR client to retrieve the element at index i from the PIR server without revealing any information about i. A PIR protocol has three procedures: PIR.GenQuery, PIR.Answer, and PIR.Decode. The PIR client first constructs a PIR query using the PIR.GenQuery method and sends it to the PIR server. This query is typically the encryption of a one-hot vector of length n, with a 1 at index i and 0 at all other places. It can also be the encryption of n 0's if no element is desired. The PIR server then runs PIR.Answer to generate a PIR response and send it back to the PIR client. The PIR client runs PIR.Decode to decode the response and retrieve the element at index i, or an empty value if no element was desired.

Pantheon uses a slightly modified version of the usual PIR protocol. The PIR query does not directly come from the Pantheon client. Recall that, if $k \in K$, the output of Pantheon's step 2 (equality check §3.4.2) is the encryption of a one-hot vector, with a 1 at index i such that $K[i] == k$, and 0 at all other places. In the case where $k \notin K$, step 2 outputs an encryption of all 0's. As a result, the output of step 2 can be readily used as the PIR query for step 3. Then, the Pantheon server runs the PIR.Answer method on the value-array V and generates a PIR response. The PIR response is then sent back to the Pantheon client. The specific PIR library and other details are discussed in the implementation section (§4).

3.4.4 ***Step 4: Decode server response.*** After receiving a response from the Pantheon server, the client decodes it to retrieve the value corresponding to k. Since the server's response is the output of a PIR.Answer method, the client can use PIR.Decode to retrieve the corresponding value. Obtaining an empty value from PIR.Decode implies that k is not present in the key-value store.

3.5 Query Compression Optimization

The output q of the query encoding procedure discussed in step 1 (§3.4.1) grows linearly in size with client's query key k. More specifically, if k is αt bits long, then q will consist of α ciphertexts. This linear growth increases the network bandwidth cost linearly, creating a scalability bottleneck. Therefore, we propose a query compression method that always reduces the size of q to one ciphertext. The Pantheon server upon receiving this compressed query, needs

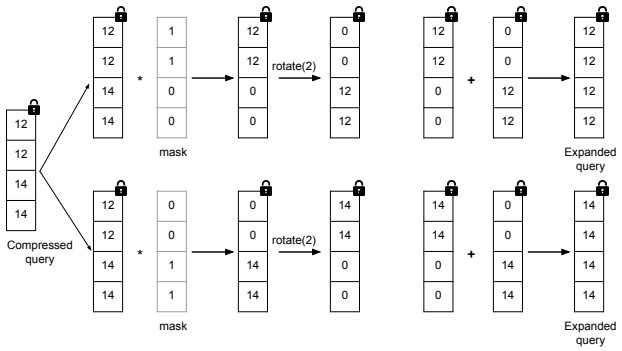


Figure 3: **Query expansion procedure at Pantheon server (assuming N = 4 and p = 17).**

to perform an expansion, so that it can obtain the α ciphertexts required to perform subsequent operations.

3.5.1 ***Compression.*** Suppose the size of each key is αt bits, where α is a factor of N and $t = (\lceil \log_2 p \rceil - 1)$. The client first splits k into α equal chunks, each of length t bits. The client also splits a plaintext vector into α equal parts, each consisting of N/α elements. Each part of this plaintext vector is then filled with the corresponding chunk of k. For instance, let $N = 4$, $p = 17$, $\alpha = 2$. The client query key is an 8-bit binary string 11001110. So, the client can split it into 2 chunks $\{1100, 1110\}$, also represented as integers $\{12, 14\}$. Now, instead of constructing two different ciphertexts for 12 and 14, the client can split the plaintext into two parts and fill them as $(12, 12, 14, 14)$ and encrypt it. This principle can be extended for any α that is not a factor of N by splitting the plaintext vector into $\alpha\prime$ equal parts, where $\alpha\prime$ is the smallest factor of N such that $\alpha\prime > \alpha$. The client then uses the first α parts of the plaintext to fill with α chunks of k and the remaining parts with 0. As a result, the client query q will always consist of a single ciphertext.

3.5.2 ***Expansion.*** The Pantheon server needs to expand the single query ciphertext into α different ciphertexts to proceed with the oblivious equality check. From the previous example, the server receives a ciphertext encrypting $(12, 12, 14, 14)$ and wants to expand it to two ciphertexts encrypting $(12, 12, 12, 12)$ and $(14, 14, 14, 14)$. The expansion procedure is demonstrated in Figure 3. First, the server constructs α plaintext vectors to mask out each of the unique values from the query ciphertext obliviously. If α is a factor of N, each mask is split into α equal parts each containing N/α slots of the plaintext vector. Otherwise, each mask is split into $\alpha\prime$ parts as used for compression. For the i^{th} mask plaintext, the i^{th} part is filled with 1 and the remaining slots with 0. Considering the previous example, the server constructs $\alpha = 2$ masks $(1, 1, 0, 0)$ and $(0, 0, 1, 1)$. The server then performs MultiplyPlain with the compressed query q and the mask plaintext $(1, 1, 0, 0)$ as input to obtain a ciphertext u that is the encryption of $(12, 12, 0, 0)$. As a result, this mask plaintext helps to filter out the value 12 from the query. The server then computes $u\prime \leftarrow \text{Rotate}(u, 2)$. So, $u\prime$ will be the encryption of $(0, 0, 12, 12)$. The server then computes $c_{add} \leftarrow \text{Add}(u, u\prime)$. This rotation and addition can be repeated for $\log_2 \alpha$ times to obtain the desired expanded query ciphertext encrypting $(12, 12, 12, 12)$. A similar process can be repeated with mask $(0, 0, 1, 1)$ to get the other expanded query encrypting $(14, 14, 14, 14)$.

3.6 Security analysis.

Pantheon provides *query privacy* (§2.3.1) to its clients. In this section, we sketch a formal proof of Pantheon's privacy guarantee. Let us define a security game $\mathcal{G}_0$ between a challenger and an adversary as defined in (§2.2). The adversary supplies two keys k^0 and k^1 of the same size and the challenger randomly selects one of them as k^b, where $b \in \{0, 1\}$. The challenger performs a query with k^b using Pantheon's protocol. Then, the adversary outputs its guess $b' \in \{0, 1\}$. The adversary wins the game if its guess is correct, i.e., $b' = b$. Let S_0 be the event that $b' = b$ in $\mathcal{G}_0$. Let us also consider another game $\mathcal{G}_1$ where the challenger simulates Pantheon's protocol with a key selected uniformly randomly from the key space, and let S_1 be the event that $b' = b$ in $\mathcal{G}_1$. Evidently, $\Pr[S_1] = 1/2$, since the challenger's query key is independent of those provided by the adversary, and the adversary can only perform random guesses. Now, in $\mathcal{G}_0$, according to Pantheon's protocol, the challenger encrypts k^b using Fully Homomorphic Encryption. Therefore, the advantage of the adversary being able to distinguish it from a random string is, $|\Pr[S_0] - \Pr[S_1]| \leq \epsilon_{FHE}$. Substituting the value of $\Pr[S_1]$ we get, $|\Pr[S_0] - 1/2| \leq \epsilon_{FHE}$. So, the adversary cannot win the game $\mathcal{G}_0$ with non-negligible advantage.

4 IMPLEMENTATION DETAILS

Our prototype of Pantheon consists of ≈3,000 lines of C++ code. It uses the homomorphic encryption functionalities provided by the state-of-the-art Microsoft SEAL [60] library. We modify the open-source implementation of Microsoft SEAL, which is a single-threaded library, to provide parallel execution over multiple cores of a machine. For the Private Information Retrieval step (§3.4.3) of Pantheon, we parallelize the open-source implementation of the FastPIR [30] library. We use FastPIR because it requires lower processing time to generate a PIR response [4] and it also uses the vectorized variant of BFV scheme, resulting in a smoother interface with the other components of Pantheon. The details of our parallel implementation of these two libraries are discussed later in this section (§4.1).

Parameter selection. Recall that the BFV scheme has three parameters: the size of the plaintext vector N, the upper bound of each plaintext element p, and the upper bound of each ciphertext element $p\prime$. N must be of the form 2^j, where j is an integer such that $10 \leq j \leq 15$ [7]. We choose N to be 2^{15}. p must be a prime number such that $p \equiv 1 \pmod{2N}$. Selection of the particular value for p needs two considerations. Since the size of each plaintext element is $t = \lceil \log_2 p \rceil - 1$ bits, larger size of p in number of bits allows a larger plaintext element, thus accommodating a bigger chunk of client's query key in a single BFV ciphertext. On the other hand, larger p increases the computational cost of raising a ciphertext to the power $(p - 1)$, required in the equality checking step (§3.4.2). Our implementation of Exponentiate uses the repeated squaring method and the number of calls to Multiply increases with the number of 1 bits in the binary representation of $(p - 1)$. For the same bit-length of p, the cost of Exponentiate is minimized if p is of the form $2^j + 1$. Therefore, we choose $p = (2^{16} + 1)$, which is a prime number congruent to $1 \pmod{2N}$. Another noteworthy point is, SEAL considers a pair of adjacent slots in a ciphertext together for its rotation operations. Therefore, it is convenient to consider two

BFV plaintext slots together to represent a chunk of the client's query key. So, each chunk of client's query key in Pantheon is $2t$ (32-bits). As a result, number of such chunks in a ciphertext becomes $N/2$ or 2^{14}. We choose $p\prime$ as a 780-bit composite number, which is generated by the Microsoft SEAL library as a product of 13 default prime numbers, each of length 60 bits. These parameters guarantee 128 bit security [7], and ensure $p\prime \gg p$ which is essential for the correct execution of Pantheon's server-side operations.

4.1 Parallelization

We parallelize the server-side operations of Pantheon to make it vertically scalable by utilizing all the cores in a multi-core machine. We divide the parallelization into two levels: 1) coarse-grain parallelization, and 2) fine-grain parallelization. In coarse-grain parallelization, we implement data parallelization by operating on different parts of the key-value store in parallel. In fine-grain parallelization, we write a wrapper on the Microsoft SEAL [60] library to parallelize the homomorphic operations it provides. We discuss both of these parallelization levels in detail.

4.1.1 ***Coarse-grain parallelization.*** The Pantheon server performs three procedures while serving a client: query expansion, equality check, and private information retrieval (PIR). In coarse-grain parallelization, we parallelize each of these three procedures. Brief details of the parallelization techniques are discussed below:

Query expansion. The server receives a compressed query ciphertext and expands it to α ciphertexts using α different masks (§3.5). The operations involving different masks are independent and can be executed in parallel. We thus parallelize the query expansion procedure by a factor of α by operating on the different masks in parallel, as shown in Figure 3.

Equality check. The server checks equality between the array of keys K and the expanded query ciphertexts. Suppose, the size of each key is αt bits, where t is the size of each element in a BFV plaintext. Then, the expanded query will consist of α ciphertexts. Also, let there be βN tuples in the key-value store, where N is the size of each BFV plaintext vector. Following the equality check step (§3.4.2), we first horizontally partition the array of keys K into β parts, each containing N keys. Then, the output of the equality check will consist of β ciphertexts, each denoting the equality between the expanded query and one of the β partitions. The Pantheon server can process each of these β partitions of K in parallel for equality check.

Now, let us focus on one such horizontal partition consisting of N keys. Each key can be split into α equal slices, each slice consisting of t bits. So, each horizontal partition is further vertically partitioned into α blocks. The i^{th} block needs to be checked for equality as shown in Figure 2 with the i^{th} query ciphertext obtained after query expansion. The Pantheon server can execute these α equality checks in parallel. However, after obtaining the results of these equality checks, they need to be multiplied together to obtain a logical *AND* of the intermediate results. So, this multiplication needs to wait for all the α intermediate equality checks to be completed.

Private Information Retrieval. The Private Information retrieval (PIR) step uses the output of the equality check and the value-array V to output an encryption of the client's desired value (§3.4.3). We parallelize the state-of-the-art FastPIR [30] library to implement this step of the Pantheon server. We vertically partition the value array into multiple parts and perform PIR on each part in parallel. Then, we conduct the "rotate and add" merging policy discussed in FastPIR [30] to merge them and obtain a consolidated PIR response.

4.1.2 ***Fine-grain parallelization.*** The coarse-grain parallelization discussed above reduces the server-side latency by processing different parts of the key-value store in parallel. However, this advantage becomes insignificant when the size of the key-value store is small and there are not enough partitions to process simultaneously. Moreover, the latency is dictated by the single threaded execution of the Exponentiate method, which in turn makes multiple sequential calls to the computationally expensive Multiply method. So, the Pantheon server may run into a situation where it has idle CPU's but the latency cannot be reduced just by applying coarse parallelization. To address this issue, we modify the BFV homomorphic operations provided by the Microsoft SEAL library to make them use multiple CPU cores in parallel. However, we cannot obtain perfect parallelization as the operations consist of parts that depend on the outcome of prior computations and must be executed in sequential order. We carefully examine the computation graphs of the relevant homomorphic operations and identify the tasks that can be executed in parallel. We primarily adopted two principles in fine grain parallelization. First, a BFV ciphertext consists of two polynomials and we execute the same operation on the two polynomials in parallel. Second, each element of a BFV ciphertext can be decomposed into a number of factors using the Chinese Reminder Theorem, and each factor can be processed in parallel for the same homomorphic operation. We use the OpenMP [53] library to implement the fine grain parallelization of Microsoft SEAL [60].

4.2 Coordinator-worker architecture

Parallelizing the server-side operations in Pantheon improves performance over a single machine. However, to retain low latency for a large key-value store containing millions of tuples, we horizontally scale Pantheon by distributing the server-side workload over a cluster of machines. The cluster deployment of the Pantheon server follows a coordinator-worker architecture. A coordinator node receives the query from the client and then distributes the value retrieval workload among a number of worker nodes. Each worker node performs its part of the computation and sends the result to the coordinator. The coordinator then aggregates the partial results produced by the workers to generate the final response and sends that back to the client.

A high-level description of the coordinator-worker arrangement is as follows. Initially, there is a cluster setup phase where the key-value store is partitioned, and each worker stores a partition in its memory. Suppose there are w workers and a total of n tuples in the key-value store. Let $n = \beta N$, where N is the size of a BFV plaintext vector (§3.2), and β is a positive integer. Then, the key-value store can be partitioned into β parts, each consisting of N tuples. A partition size cannot be smaller than N because that is the smallest granularity upon which a BFV plaintext and ciphertext can operate. Then, the β partitions can be distributed among the w workers, each worker containing at most $\lceil \beta/w \rceil$ such partitions, i.e., a disjoint subset of $\lceil (\beta N/w) \rceil$ tuples from the key-value store.

After the cluster setup is complete, the Pantheon server can serve queries from its clients. A client sends its compressed query q to the coordinator. The coordinator broadcasts q to all w workers. Each worker performs the query expansion operation (§3.5) to obtain the expanded query. Then each worker executes the equality check (§3.4.2) and Private Information Retrieval (PIR) (§3.4.3) steps on its partition of the key and value arrays, respectively. Since at most one of the workers can have the desired key in its key array, that worker will generate the client's desired PIR response. Each of the other workers will generate a PIR response of an empty value. All workers send their partial responses back to the coordinator. PIR responses have an additive homomorphic property such that adding empty responses to a response encrypting the desired value yields a new response that also encrypts the same desired value. Similarly, if all the w workers generate PIR responses of empty value, adding them together yields a new PIR response of empty value. The coordinator performs this aggregation by adding the PIR responses from all w workers and sends the aggregated response back to the client.

5 EVALUATION

Our evaluation focuses on Pantheon's server-side latency for serving private GET queries. We compare Pantheon's performance with Constant-weight Keyword PIR (CKP) [46], the state-of-the-art system for private key-value retrieval. First, we provide a microbenchmark of Pantheon's and CKP's different components and analytically explain the reasons for Pantheon's performance superiority over CKP. Then, we compare the performance of the parallelized implementation of Pantheon (§4.1) with the open-source multi-threaded implementation [24] of CKP over all the cores of a single AWS instance. After that, we demonstrate the effect of Pantheon's parallelization techniques (§4.1) by comparing the final version with two other intermediate versions: Pantheon (S) - a single-threaded implementation with no parallelization, and Pantheon (C) - a version containing only the coarse-grain parallelization. We configure all the variants of Pantheon and the Constant-weight Keyword PIR system to provide 128-bits of security [7].

Experiment setup. We run all our experiments in AWS EC2 US East region (Ohio). The single-machine experiments use one instance of type `c5.12xlarge` (48 VCPU, 96 GiB of RAM, and 12 Gbps of network bandwidth) as the server. For Pantheon's cluster deployment, we use one instance of type `c5.24xlarge` (96 VCPU, 192 GiB of RAM, and 25 Gbps of network bandwidth) as the coordinator and 128 instances of type `c5.12xlarge` as workers. For each experimental configuration, we generate the keys by taking the hash digest of integers in $\{1, 2, ..., n\}$ to ensure unique keys. The values are filled in with random bit-strings. We repeat each experiment 10 times, discard the minimum and maximum values to avoid outliers, and then take the average of the remaining values.

5.1 Microbenchmarks

We benchmark both Pantheon and Constant-weight Keyword PIR (CKP) [46] on a single core of an AWS `c5.12xlarge` instance with two different key sizes. We run Pantheon with 32-bit and 64-bit keys. Since CKP does not support key size larger than 60 bits, we run it with 32-bit and 60-bit keys. CKP has a configurable parameter named hamming-weight of the keys (denoted as h). We consider $h = 3$ for 32-bit key configuration and $h = 5$ for 60-bit key configuration. These values of h yield the minimum latency for the respective key sizes. For all the configurations, we take the number of tuples in the key-value store (n) as 16,384 and the size of each value as 256 bytes. Table 1 shows the server-side cost for each of the three steps in the execution of a GET query: 1) Query expansion, 2) Equality check, and 3) Private Information Retrieval (PIR). We present an analytical discussion of the component-wise cost as follows.

Table 1: **Microbenchmarks for different operations to serve a GET query by both Pantheon and Constant-weight Keyword PIR (CKP). All the configurations use 16,384 tuples in the key-value store with each value being 256 bytes.**

	32-bit key		64-bit key	
	CKP	Pantheon	CKP	Pantheon
Total server time (sec)	**463.38**	**3.64**	**921.2**	**6.90**
Query expansion (sec)	12.92	0	39.98	0.15
Equality check (sec)	438.36	3.08	869.12	6.19
PIR (sec)	12.10	0.56	12.10	0.56
Number of operations				
Substitution/ Rotation	4095	0	12286	2
MULTIPLY	32768	16	65536	33

Query expansion. CKP has substantially higher query expansion time compared to Pantheon. This is because, in CKP, a single query ciphertext needs to be expanded to m ciphertexts such that, $\binom{m}{h} \geq 2^{keysize}$. The expansion procedure makes repeated calls to an expensive homomorphic substitution operation. On the other hand, in Pantheon, one query ciphertext is expanded to α ciphertexts where $\alpha = \lceil \frac{keysize}{2\log_2 p} \rceil$. This involves $\alpha\lceil \log_2 \alpha \rceil$ calls to expensive rotation operations which is analogous to the substitution operation used in CKP. For example, with 60-bit keys, a query ciphertext in CKP is expanded to 10,673 ciphertexts with a total size of ≈5.21 GiB, involving 12,286 substitution operations. The total time for this expansion step is 39.98 seconds. On the other hand, a query in Pantheon for 64-bit keys expands to 2 ciphertexts with a total size of 6 MiB, requiring 2 rotation operations and a total time of 0.15 seconds. A single rotation operation in Pantheon takes more time than a substitution in CKP due to its larger parameters, but the total time for all the operations is significantly lower.

Equality check. The Equality check is the most expensive step for both Pantheon and CKP. The cost of the equality check is dominated by the BFV MULTIPLY operation to multiply two ciphertexts together. CKP requires $n(h-1)$ total number of calls to MULTIPLY, where n is the total number of tuples in the key-value store and h is the hamming-weight of each key, a tunable parameter for CKP. On the other hand, Pantheon requires $(\alpha \cdot \frac{n}{N/2} \cdot \log_2 p + (\alpha - 1))$ multiplications, where $\alpha = \lceil \frac{keysize}{2\log_2 p} \rceil$. Even though the asymptotic relation with number of elements n is the same, the $\frac{1}{N}$ factor in Pantheon's cost significantly reduces the number of ciphertext multiplications. For instance, as shown in Table 1, for 32-bit keys the number of multiplications in CKP is 32,768, whereas that in Pantheon is only 16. The corresponding times required for equality check are 438.36 and 3.08 seconds respectively. Note that, due to the larger parameter size, a single MULTIPLY operation in Pantheon takes ≈14× more time than that in CKP. Even then, due to the significantly smaller number of multiplications, the overall time for equality check is smaller in Pantheon.

Private information retrieval. Pantheon also takes less time than CKP for the PIR step. The performance gain is mainly due to the performance difference between FastPIR [4] (used by Pantheon) and SealPIR [9] (used by CKP). It is worth mentioning that the design of the equality operator in Pantheon allows it to take advantage of FastPIR, which performs significantly faster for smaller objects. On the other hand, the output of CKP's equality operator does not allow it to utilize the plaintext packing optimization provided by SealPIR, thus requiring more computation in the PIR step. However, as objects grow bigger, the difference between the PIR cost of Pantheon and CKP gets smaller (§5.2). A more comprehensive comparison between FastPIR and SealPIR is available in [4].

5.2 Single-machine latency

Factors affecting Pantheon's latency. We conduct experiments varying three size-related parameters: the number of tuples in the key-value store (n), the size of each key, and the size of each value. Pantheon's server-side latency for a GET query comprises the processing time for three operations: i) query expansion (§3.5), ii) equality check (§3.4.2), and iii) private information retrieval (§3.4.3). We briefly discuss how the three parameters affect the processing time for each operation. The processing time for query expansion depends on the number of ciphertexts obtained after expansion. Therefore, this time increases linearly with the size of each key but is independent of the number of tuples (n) and the size of each value. The processing time for the equality check depends on the total size of the array of keys K. Therefore, this time increases linearly both with the size of each key and the number of keys in K, but is independent of the size of each value. Finally, the processing time for PIR depends on the size of the value-array V and is independent of the size of each key. As explained in FastPIR [4], Pantheon's PIR time increases linearly with the size of each value, and linearly with a slope smaller than 1 with the number of tuples. These relationships will be reflected in our experimental results discussed later.

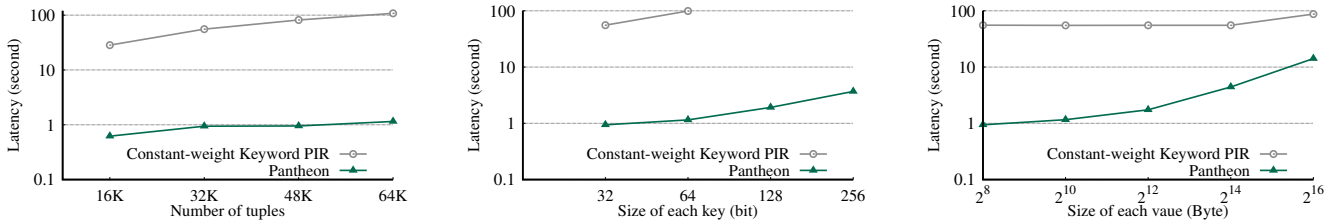
Varying the number of tuples. Figure 4a shows how the server-side latency of both Pantheon and Constant-weight Keyword PIR (CKP) depend on the number of tuples (n) in the key-value store. We vary the number of tuples from $16,384$ to $65,536$ while keeping the size of each key 32 bits and each value 256 bytes. In general, Pantheon's latency is much lower than that of CKP. For instance, when $n = 65,536$, CKP's server-side latency for serving one GET query is 107.8 seconds, whereas the latency for Pantheon is 1.15 seconds, a 93× improvement. Also, the baseline latency increases almost linearly from 28.4 seconds for $n = 16,384$ to 107.8 seconds for $n = 65,536$. On the other hand, Pantheon's latency grows from 0.62 seconds for $n = 16,384$ to 1.15 seconds for $n = 65,536$, an increase of only 1.85×, whereas the number of tuples (n) increases 4×. This slower growth of latency is because the time for query expansion remains constant as the number of tuples increases, and the time for FastPIR increases with a slope smaller than 1.

Varying the key size. Figure 4b shows how the latency for Pantheon and the baseline change with the size of each key. For Pantheon, we vary the size of each key from 32-bit to 256-bit. However, the Constant-weight Keyword PIR implementation does not support key-size larger than 60-bit. So, we run CKP for 32-bit and 60-bit keys. We take $32,768$ tuples in the key-value store for both systems, with each value being 256 bytes. For 32-bit keys, the latency for Pantheon is 0.94 seconds, 59× better than that of the baseline latency of 55.48 seconds. Pantheon's latency increases to 3.69 seconds for 256-bit keys.

Varying the value size. We populate the key-value store with $n = 32,768$ tuples while keeping the size of each key 32-bits. We vary the size of each value from 256 bytes to $65,536$ bytes. Figure 4c shows how the latency of Pantheon and CKP vary with the size of each value. Pantheon's latency for 256-byte values is 0.94 seconds and increases to 1.74 seconds for $4,096$-byte values, and then to 13.28 seconds for $65,536$-byte values. Only the private information retrieval (PIR) step at the Pantheon server is affected by the size of values. The PIR processing time increases linearly while the processing times for query expansion and equality check remain unchanged with the size of values. Therefore, for Pantheon, variation in the size of value primarily reflects the performance of FastPIR. The baseline latency with the variation of the value-size remains constant at 55.4 seconds for values up to $16,384$ bytes and then increases to 87.47 seconds when the values are $65,536$ bytes. This is because CKP's PIR computation increases following a step function with every 20KB increase in value-size.

Effect of parallelization. We apply two levels of parallelization on the single-threaded implementation Pantheon-(S). First, we apply coarse-grain parallelization on Pantheon-(S) to obtain Pantheon-(C). Then, we apply fine-grain parallelization on top of Pantheon-(C) to get the final version of Pantheon. Figure 5 shows the impact of these two levels of parallelization on a key-value store containing $32,768$ tuples with 64-bit keys and 256-byte values. Evidently, the equality check takes the bulk of the server-side processing time. For coarse-grain parallelization, we horizontally partition the key array into two parts, each containing $16,384$ tuples. We further split each key into two slices, each of 32-bits. This is the smallest partition size ($16,384$-tuples by 32-bits) allowed by Pantheon's BFV parameters (§4). Coarse-grain parallelization reduces the equality check time from 11.85 seconds to 3.19 seconds, an improvement of 3.7×. Our fine-grain parallelization of the Microsoft SEAL library further reduces the equality check time by a factor of 1.9× to 1.71 seconds. Recall that only a subset of the computations in Microsoft SEAL [60] can be executed in parallel for fine-grain parallelization. Hence, the sequential parts are the bottleneck for reducing the latency further.

Figure 6a shows the performance gains due to parallelization when the number of tuples in the key-value store equals the minimum horizontal partition size of $16,384$. We vary the size of each key from 32 bits to 256 bits. The performance gain of Pantheon-(C) over Pantheon-(S) depends on the number of partitions that the key-value store can be divided into. For 32-bit keys, Pantheon-(C) cannot partition the key-array, so the latency is 3.15 seconds, which is very close to the Pantheon-(S) latency of 3.5 seconds. The parallelization of the PIR operation reduces the latency by 0.35 seconds. The total latency, in this case, is dominated by the 2.98 seconds required for the equality check operation, which cannot be reduced further by coarse-grain parallelization. We use fine-grain parallelization of the Microsoft SEAL library to reduce the processing time of the equality check operation to 0.55 seconds. The total latency obtained by fine-grain parallelization for 32-bit keys is 0.62 seconds, an improvement



(a) **Variation of latency with number of tuples. The size of each key is 32 bits, and each value is 256 bytes.**

(b) **Variation of latency with key-size. CKP does not support keys larger than 60 bits, and so is shown for 32-bit and 60-bit keys. The key-value store contains 32,768 tuples, where each value is 256 bytes.**

(c) **Variation of latency with value-size. The key-value store contains 32,768 tuples, where each key is 32 bits.**

Figure 4: **Latency incurred by a single-machine server to perform a GET query with the variation of three size related parameters.**

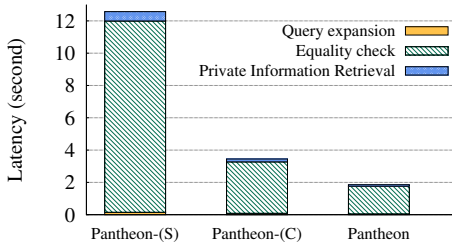
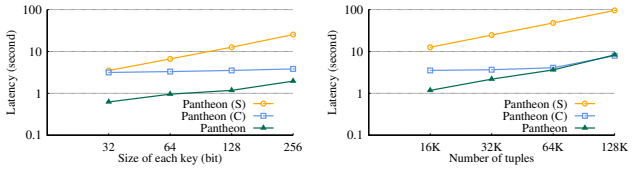


Figure 5: **Impact of the two levels of parallelization on Pantheon's three server-side operations. The key-value store has 32,768 tuples, with 64 bit keys and 256 byte values. Pantheon-(S) is the single-threaded implementation with no parallelization, Pantheon-(C) includes only coarse-grain parallelization, and Pantheon is the final version containing both coarse-grain and fine-grain parallelization.**



(a) **Variation of latency with key-size, number of tuples fixed at 16,384.**

(b) **Variation of latency with number of tuples, key-size fixed at 128 bits.**

Figure 6: **Impact of parallelization on single machine server-side latency. Pantheon-(S) is the single-threaded implementation with no parallelization, Pantheon-(C) includes only coarse-grain parallelization, and Pantheon is the final version containing both coarse-grain and fine-grain parallelization. In all cases, the value-size is 256 bytes.**

of 5× over Pantheon-(C). As the key-size increases, Pantheon-(C) can partition the key-array into multiple parts and use multiple cores in parallel for the equality check operation. So, the latency of Pantheon-(C) grows slowly to 3.83 seconds for 256-bit keys, and its difference with the final version of Pantheon reduces gradually.

Figure 6b shows the impact of parallelization on a heavier workload. We keep the key-size at 128-bit, value-size at 256-byte, and vary the number of tuples from 16, 384 to 131, 072. Pantheon-(C)'s latency for 16, 384 tuples is 3.53 seconds and increases slowly to 4.08 seconds for 65, 536 tuples. After that, the latency increases to 7.91 seconds for 131, 072 tuples. We speculate that the machine gets saturated at this point. On the other hand, Pantheon's implementation with both the coarse-grain and fine-grain parallelization has a latency of 1.18 seconds for 16, 384 tuples and 3.63 seconds for 65, 536 tuples, lower than the corresponding latencies for Pantheon-(C). However, Pantheon's latency for 131, 072 tuples is 8.29 seconds, marginally higher than Pantheon-(C). So, once the machine gets

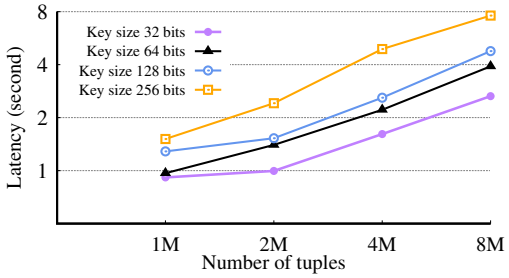


Figure 7: **Server-side latency with variations in the number of tuples when the Pantheon server is deployed over a cluster of 128 worker machines. Each curve shows the latency trend for a different key-size, while the value-size is fixed at 256 bytes.**

saturated for Pantheon-(C), the overhead of fine-grain parallelization makes the performance of Pantheon worse. This result imposes an interesting design decision. If low latency is the primary goal, then the final version of Pantheon is preferable by distributing Pantheon's workload over a large number of machines while keeping the partition size at each machine smaller. On the other hand, if reducing CPU cost is of primary interest, Pantheon-(C) may yield a better result with larger partitions over fewer machines.

5.3 Cluster latency

Figure 7 shows Pantheon's latency trend when its server is distributed over a cluster of 128 worker instances. We vary the number of tuples from $1M$ to $8M$, the size of each key from 32-bit to 256-bit, and keep the size of each value at 256 bytes. Pantheon's final version containing both coarse-grain and fine-grain parallelization gives the lowest latency for all these experimental configurations. For the cluster deployment, the server-side latency consists of two components: i) processing time at the worker and ii) coordination overhead. The processing time depends on the size of the partition processed by each worker. The coordination overhead comprises the time required to broadcast the compressed query ciphertext to all the workers and the time to aggregate the PIR responses from each worker. The coordination overhead is fixed for a particular cluster configuration because the query size and the PIR response size from each worker remain the same irrespective of the size of the key-value store. Pantheon's server-side latency increases with both the number of tuples and the size of each key. For example, for $2M$ tuples, the latency for 32-bit keys is 0.99 seconds and increases to 2.42 seconds for 256-bit keys. The corresponding processing times are 0.69 seconds and 2.12 seconds, respectively, while the coordination overhead for both is 0.3 seconds. If the number of tuples

is increased to $8M$ while keeping the key-size at 256-bit, the latency rises to 7.6 seconds. Out of this, 7.3 seconds are for processing, and the coordination overhead is 0.3 seconds. The processing time increases by 3.4×, slightly slower than the 4× increase in the number of tuples, mainly because of a slower growth of PIR time (§5.2).

5.4 Resource overheads of Pantheon

This section discusses the overheads Pantheon imposes on its clients and estimates the dollar cost of a private Get query.

Client-side overhead. The client-side overhead of Pantheon stays fixed irrespective of the size of the key-value store. A Pantheon client incurs a cpu time of 0.07 seconds for retrieving a value privately from the Pantheon server. This cpu time comprises the time for encrypting a compressed query and decoding the PIR response from the server. A Pantheon client uploads 3MiB for each query and downloads 1.5MiB for each response.

Dollar cost. We convert the server-side cpu time and network usage for performing a private Get query to a dollar amount. Amazon EC2 charges $0.744 per hour for each worker instance (`c5.12xlarge`) and $1.488 per hour for the coordinator instance (`c5.24xlarge`) [62]. We use these unit costs to calculate the total cost for 128 workers and a coordinator for the duration of serving a query. For network usage, we use Amazon's pricing model of $0.05 per GiB download (Amazon does not charge for uploads) [61]. For Pantheon, the cost for performing a private Get query over a key-value store containing $1M$ tuples, where each key is 256-bit, and each value is 256-byte, sums up to 4 cents. This cost rises to 20 cents when there are $8M$ tuples in the key-value store.

6 RELATED WORK

Querying over private data. There is a large body of work in the literature that allows a client to outsource its private data and perform queries on it. These works require the client to encrypt their data using their secret keys and store the encrypted data in the cloud server. The server then uses different techniques to serve the client's query while preserving privacy. CryptDB [55] allows a client to encrypt its data using multiple layers of encryption (onion encryption) and supports a subset of SQL queries over the encrypted data. Other works use techniques such as order preserving encryption [1], homomorphic encryption [58, 68], and searchable encryption [19] to hide clients' data. Several research works [14, 41, 56, 66, 70] and industry deployments [11–13, 63] provide encrypted database service using Trusted Execution Environment (TEE), such as Intel SGX [25]. In this setup, the client stores the encrypted data in the server and shares the secret key with trusted hardware located at the server. The trusted hardware decrypts the data while executing a client query. However, these works do not hide the access pattern of the client's query and are vulnerable to inference attacks that can retrieve the data in its plaintext form [15, 37, 38, 43, 51]. Arx [54] uses a combination of homomorphic encryption and garbled circuit to hide query access pattern over encrypted data. Oblidb [28] and Oblix [48] hide query access pattern over private data using TEE and ORAM [33]. PANCAKE [36] uses frequency smoothing to hide query access pattern on a private key-value store. Pantheon's problem domain is different from these works since Pantheon deals with *public* data, and the client cannot encrypt the data as required in these solutions. Conceptually, one may develop a solution to hide access pattern over a public key-value store using TEE [47], where a client encrypts the query and the server decrypts it inside a trusted enclave. However, this approach guarantees a weaker privacy since TEEs are susceptible to side-channel attacks [44, 59, 64, 65], cache attacks [18, 26, 34], and fault injection attacks [20, 50] that can reveal the client's secret key and thus break the privacy.

Querying over public data The problem of searching privately over public data falls in the domain of Private Information Retrieval (PIR) [22, 23, 42]. In its basic form, PIR allows a client to obliviously retrieve the element at a particular index of a data array located at an untrusted server. Existing applications of PIR [2–4, 10, 35, 39, 45, 49] work in a setting where the client knows the index of the desired element in advance. Pantheon works in a different setting. A Pantheon client does not know whether the desired key exists in the key-value store, let alone its index. An extension of PIR, known as keyword PIR [21], allows a client to retrieve an element corresponding to a keyword using $\lceil \log_2(n+1) \rceil + 1$ round-trip interactions between the client and the server [10]. However, this technique deviates from Pantheon's goal of single round value retrieval (§2.3.2). The primary challenge for retrieving value in a single round is checking the equality between two keys. A body of work [5, 6, 27, 32] uses Fully Homomorphic Encryption (FHE) [16, 31] to implement an oblivious equality check operator between keys and can perform value retrieval in a single round. However, they involve prohibitively expensive operations and thus incur very high latency. A recent work by Mahdavi et al. [46] improves over the existing FHE-based equality checking by devising a new equality operator named constant-weight equality operator. This equality operator requires mapping each key to a constant-weight codeword, i.e., binary strings containing the same number of 1's. They use this equality operator and the SealPIR [9] library to develop a single-round solution for the private key-value retrieval problem. Pantheon's approach aligns with this work. Pantheon uses a faster equality check operator that relies on Fermat's little theorem [57] and FastPIR [30] for better performance.

7 CONCLUSION

Providing privacy for clients querying a *public* key-value store is challenging because clients have no control over the data. Therefore, the access pattern of a client's query must be hidden from the server to guarantee privacy. Prior work that provides query privacy to a client either requires multiple rounds for each retrieval or has performance and scalability bottlenecks. This paper presents Pantheon, a round-optimal solution for private retrieval from a public key-value store, that scales to millions of tuples. When the Pantheon server is deployed over a cluster of 128 machines with a key-value store containing 2M tuples, where each key is 32 bits and each value is 256 bytes, the server-side latency for serving a client's query privately is under one second. Pantheon, for the first time, shows that it is possible to provide strong privacy for querying over a practically large public key-value store with reasonable latency.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This work is funded in part by NSF grants CNS-1703560 and OAC-2126327.

REFERENCES

[1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 563–574.

[2] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. 2016. XPIR: Private Information Retrieval for Everyone. In *Privacy Enhancing Technologies Symposium (PETS)*. 155–174.

[3] Ishtiyaque Ahmad, Laboni Sarker, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. 2021. Coeus: A System for Oblivious Document Ranking and Retrieval. In *ACM Symposium on Operating Systems Principles (SOSP)*. 672–690.

[4] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. 2021. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 313–329.

[5] Adi Akavia, Dan Feldman, and Hayim Shaul. 2019. Secure data retrieval on the cloud: Homomorphic encryption meets coresets. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), 80–106.

[6] Adi Akavia, Craig Gentry, Shai Halevi, and Max Leibovich. 2019. Setup-Free Secure Search on Encrypted Data: Faster and Post-Processing Free. *Proceedings on Privacy Enhancing Technologies* 3 (2019), 87–107.

[7] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, et al. 2021. Homomorphic encryption standard. In *Protecting Privacy through Homomorphic Encryption*. Springer, 31–62.

[8] Josh Allen. 2021. Recent Cyber Attacks & Data Breaches In 2021. https://purplesec.us/recent-cyber-security-attacks/ 2022-12-14.

[9] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. 2018. PIR with compressed queries and amortized query processing. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 962–979.

[10] Sebastian Angel and Srinath Setty. 2016. Unobservable communication over fully untrusted infrastructure. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 551–569.

[11] Panagiotis Antonopoulos, Arvind Arasu, Kunal D Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, et al. 2020. Azure SQL database always encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1511–1525.

[12] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security with Cipherbase. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[13] Arvind Arasu, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. 2015. Transaction processing on confidential data using cipherbase. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 435–446.

[14] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *USENIX Conference on File and Storage Technologies (FAST)*. 173–190.

[15] Vincent Bindschaedler, Paul Grubbs, Cornell Tech, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. 2018. The Tao of Inference in Privacy-Protected Databases. *Proceedings of the VLDB Endowment* 11, 5 (2018), 1715–1728.

[16] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual Cryptology Conference*. Springer, 868–886.

[17] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[18] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies*.

[19] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *Network and Distributed System Security Symposium (NDSS)*. 1–16.

[20] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. 2021. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In *USENIX Security Symposium (SEC)*. 699–716.

[21] Benny Chor, Niv Gilboa, and Moni Naor. 1998. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003. https://eprint.iacr.org/1998/003.

[22] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. 1995. Private Information Retrieval. In *Symposium on Foundations of Computer Science (FOCS)*. 41–50.

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

[24] Constant-weight PIR 2022. Constant-weight PIR v0.1. https://github.com/RasoulAM/constant-weight-pir.

[25] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016). https://eprint.iacr.org/2016/086.pdf

[26] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. 2018. Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2 (2018), 171–191.

[27] Yarkın Doröz, Berk Sunar, and Ghaith Hammouri. 2014. Bandwidth efficient PIR from NTRU. In *International Conference on Financial Cryptography and Data Security*. Springer, 195–207.

[28] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proceedings of the VLDB Endowment* 13, 2 (2019), 169–183.

[29] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive* (2012). https://eprint.iacr.org/2012/144.pdf

[30] FastPIR 2021. An efficient computational private information retrieval (CPIR) library. https://github.com/ishtiyaque/FastPIR.

[31] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices.. In *ACM Symposium on Theory of Computing (STOC)*. 169–178.

[32] Craig Gentry and Shai Halevi. 2019. Compressible FHE with applications to PIR. In *Theory of Cryptography Conference*. Springer, 438–464.

[33] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[34] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.

[35] Matthew Green, Watson Ladd, and Ian Miers. 2016. A protocol for privately reporting ad impressions at scale. In *ACM Conference on Computer and Communications Security (CCS)*. 1591–1601.

[36] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. 2020. PANCAKE: Frequency smoothing for encrypted data stores. In *USENIX Security Symposium (SEC)*. 2451–2468.

[37] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2019. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1067–1083.

[38] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. 2017. Leakage-abuse attacks against order-revealing encryption. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 655–672.

[39] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. 2016. Scalable and private media consumption with Popcorn. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 91–107.

[40] HElib 2021. HElib v2.0.0 Release. https://github.com/homenc/HElib/tree/master/examples/BGV_country_db_lookup/.

[41] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. Shieldstore: Shielded in-memory key-value storage with SGX. In *ACM European Conference on Computer Systems (EuroSys)*. 1–15.

[42] Eyal Kushilevitz and Rafail Ostrovsky. 1997. Replication is not needed: Single database, computationally-private information retrieval. In *Symposium on Foundations of Computer Science (FOCS)*. 364–373.

[43] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2018. Improved reconstruction attacks on encrypted data using range query leakage. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 297–314.

[44] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. 2017. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security Symposium (SEC)*. 523–539.

[45] Jilan Lin, Ling Liang, Zheng Qu, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan Xie. 2022. INSPIRE: In-Storage Private Information Retrieval via Protocol and Architecture Co-design. In *International Conference on Computer Architecture (ISCA)*. 102–115.

[46] Rasoul Akhavan Mahdavi and Florian Kerschbaum. 2022. Constant-weight PIR: Single-round Keyword PIR via Constant-weight Equality Operators. In *USENIX Security Symposium (SEC)*. 1723–1740.

[47] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. 2019. BITE: Bitcoin lightweight client privacy using trusted execution. In *USENIX Security Symposium (SEC)*. 783–800.

[48] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An efficient oblivious search index. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 279–296.

[49] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. 2011. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *USENIX Security Symposium (SEC)*. 31–46.

[50] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1466–1482.

[51] Muhammad Naveed, Seny Kamara, and Charles V Wright. 2015. Inference attacks on property-preserving encrypted databases. In *ACM Conference on Computer and Communications Security (CCS)*. 644–655.

[52] National Institute of Standards and Technology (NIST). 2015. Secure Hash Standard (SHS). (August 2015). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[53] OpenMP 2021. OpenMP API 5.2. https://github.com/OpenMP.

[54] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2019. Arx: an encrypted database using semantically secure encryption. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1664–1678.

[55] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles (SOSP)*. 85–100.

[56] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 264–278.

[57] Kenneth H Rosen. 2011. *Elementary number theory*.

[58] Stephen Tu M Frans Kaashoek Samuel and Madden Nickolai Zeldovich. 2013. Processing Analytical Queries over Encrypted Data. *Proceedings of the VLDB Endowment* 6, 5 (2013), 289–300.

[59] Michael Schwarz and Daniel Gruss. 2020. How trusted execution environments fuel research on microarchitectural attacks. *IEEE Security & Privacy* 18, 5 (2020), 18–27.

[60] SEAL 2021. Microsoft SEAL (release 3.7). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[61] Amazon Web Services. 2022. Amazon EC2 On-Demand Pricing (Data transfer). https://aws.amazon.com/ec2/pricing/on-demand/.

[62] Amazon Web Services. 2022. Amazon EC2 Reserved Instances Pricing. https://aws.amazon.com/ec2/pricing/reserved-instances/pricing/.

[63] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building enclave-native storage engines for practical encrypted databases. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1019–1032.

[64] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. 1–6.

[65] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium (SEC)*. 1041–1056.

[66] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 370–388.

[67] Ivana Vojinovic. 2022. Data Breach Statistics That Will Make You Think Twice Before Filling Out an Online Form. https://dataprot.net/statistics/data-breach-statistics/

[68] Wai Kit Wong, Ben Kao, David Wai Lok Cheung, Rongbin Li, and Siu Ming Yiu. 2014. Secure query processing with data interoperability in a cloud database environment. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1395–1406.

[69] Karl Wüst, Sinisa Matetic, Moritz Schneider, Ian Miers, Kari Kostiainen, and Srdjan Čapkun. 2019. Zlite: Lightweight clients for shielded zcash transactions using trusted execution. In *International Conference on Financial Cryptography and Data Security*. Springer, 179–198.

[70] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 283–298.