Efficient Confidentiality-Preserving Data Analytics over Symmetrically Encrypted Datasets

Savvas Savvides Purdue University savvas@purdue.edu Darshika Khandelwal Università della Svizzera italiana (USI) khandd@usi.ch Patrick Eugster Università della Svizzera italiana (USI) eugstp@usi.ch

ABSTRACT

In the past decade, cloud computing has emerged as an economical and practical alternative to in-house datacenters. But due to security concerns, many enterprises are still averse to adopting third party clouds. To mitigate these concerns, several authors have proposed to use partially homomorphic encryption (PHE) to achieve practical levels of confidentiality while enabling computations in the cloud. However, these approaches are either not performant or not versatile enough. We present two novel PHE schemes, an additive and a multiplicative homomorphic encryption scheme, which, unlike previous schemes, are symmetric. We prove the security of our schemes and show they are more efficient than state-of-the-art asymmetric PHE schemes, without compromising the expressiveness of homomorphic operations they support. The main intuition behind our schemes is to trade strict ciphertext compactness for good "relative" compactness in practice, while in turn reaping improved performance. We build a prototype system called Symmetria that uses our proposed schemes and demonstrate its performance improvements over previous work. Symmetria achieves up to $7 \times$ average speedups on standard benchmarks compared to asymmetric PHE-based systems.

PVLDB Reference Format:

Savvas Savvides, Darshika Khandelwal, and Patrick Eugster. Efficient Confidentiality-Preserving Data Analytics over Symmetrically Encrypted Datasets. *PVLDB*, 13(8): 1290-1303, 2020. DOI: https://doi.org/10.14778/3389133.3389144

1. INTRODUCTION

Cloud computing has become ubiquitous due to its economical and practical paradigm. Third-party clouds are nowadays used by both corporations and governments to perform cost-effective computations. Oftentimes, these computations require sensitive data to be moved to the cloud which places trust on the cloud provider. Resource sharing among multiple tenants only adds to the problem of ensuring

Proceedings of the VLDB Endowment, Vol. 13, No. 8 ISSN 2150-8097.

DOI: https://doi.org/10.14778/3389133.3389144

data confidentiality. Due to this fact, many organizations are still reluctant to use third party clouds.

One approach being pursued to alleviate the confidentiality concerns is *homomorphic encryption*. Fully homomorphic encryption (FHE) allows arbitrary operations to be performed directly over encrypted data and thus preserves the confidentiality of data throughout computations. Gentry introduced FHE and a first cryptosystem [16, 17] that provably achieves it. Though FHE has been becoming more practical [28], it still exhibits performance costs that are prohibitive for many computations. An alternative to FHE is partially homomorphic encryption (PHE). PHE denotes schemes that allow individual operations over encrypted data, e.g., addition, multiplication. By using multiple PHE schemes side-by-side, many operations can be supported with practical overhead. This is particularly true when property-preserving encryption (PPE) schemes are used, in addition to PHE. Ciphertexts of PPE schemes preserve some property of the underlying plaintext and can be used to carry out comparisons.

Although PHE is much faster compared to FHE, its overhead remains non-trivial. This is because most existing PHE schemes such as ElGamal [12], Benaloh [14], Paillier [33], and RSA [38] are asymmetric. Asymmetric schemes use a public key to encrypt messages and another private key to decrypt messages and tend to fall behind symmetric schemes in terms of performance since they have large ciphertext spaces leading to large ciphertext size overheads. Furthermore, the homomorphic operations of asymmetric schemes require complex computations involving arbitrary-precision arithmetic operations.

Previous attempts to propose symmetric PHE schemes that are more performant than their asymmetric counterparts have done so at the expense of *expressiveness*. The additively symmetric homomorphic encryption scheme (ASHE) [34] for example is a symmetric additive homomorphic encryption (AHE) scheme that enables addition of two encrypted values. ASHE is much faster than asymmetric AHE schemes such as Paillier but has limited expressiveness. Specifically, ASHE only supports addition of two ciphertexts, whereas other, asymmetric, AHE schemes support addition and subtraction between two ciphertexts or between a ciphertext and a plaintext value, negation of a ciphertext, as well as multiplication between a ciphertext and a plaintext.

The paucity of PHE schemes that are both performant (symmetric) and expressive in terms of the homomorphic operations they support, forced previous PHE-based systems to utilize multiple schemes for the same type of homomorphism. For example, Cuttlefish [39] switches between the symmetric,

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

more performant, ASHE scheme when only additions are required, and falls back to the asymmetric, more computationally expensive, Paillier scheme when additional homomorphic computations (such as the ones described above) are required. Similarly, Cipherbase [1] uses a dual hardware and software co-design, switching between PHE schemes and a trusted co-processor for secure computations. While switching between schemes allows these systems to execute queries more efficiently, having to utilize multiple schemes makes their design less flexible due to switching overhead.

In this paper, we thus introduce two novel symmetric encryption schemes (a) symmetric additive homomorphic encryption (SAHE) and (b) symmetric multiplicative homomorphic encryption (SMHE) designed specifically to retain the expressiveness of state-of-the-art PHE schemes while providing improved performance compared to previously used asymmetric PHE schemes. The main intuition behind the design of our schemes is to trade ciphertext compactness (size remains the same as homomorphic operations are performed) in the strict sense (i.e., qualitatively) for good "relative" compactness in practice (quantitatively), while in turn reaping improved performance in practice. We observe that:

- 1. In many scenarios ciphertext compactness (or lack thereof) has little to no effect on performance. Many applications require arithmetic operations between a few, fixed number of ciphertext values in the case of relational data, one can simply think of computations across *columns* for individual rows. Here the ciphertext size overhead added due to the lack of compactness guarantee is bounded.
- 2. By applying a set of compaction techniques (see § 3.4) we can limit the ciphertext expansion in unbounded sequences of operations such as aggregations over à priori unbounded sets of values. Examples include aggregations across *rows* in relational data. As we show empirically in § 6, our schemes achieve practical overhead despite not being compact.
- 3. Many of the homomorphic operations our schemes support, by their nature, do not increase the ciphertext size.

We provide formal proofs of security of our schemes and show that they satisfy standard notions of security, i.e., semantic security under chosen plaintext attack (CPA). To the best of our knowledge, there is no symmetric AHE scheme that preserves the expressiveness of existing asymmetric AHE schemes and there is no symmetric multiplicative homomorphic encryption (MHE) scheme at all. Here we make a distinction between PHE and somewhat homomorphic encryption (SWHE) [6] schemes. Symmetric SWHE schemess that support multiplication do exist but, unlike our schemes, only allow a limited number of operations.

We present a prototype system called Symmetria¹ that uses our proposed schemes SAHE and SMHE to perform arithmetic operations over encrypted data. We further extend Symmetria to utilize a number of existing PPE schemes to allow comparisons over encrypted data and therefore support a wider range of queries. We compare our symmetric schemes against asymmetric schemes such as Paillier and ElGamal. We evaluate Symmetria and demonstrate its low overhead over plaintext computations. Symmetria achieves average speedups of $3.8 \times$ and $7 \times$ over state-of-the-art asymmetric PHE-based systems on the standard TPC-H and TPC-DS benchmarks respectively. In summary, in this paper we make the following contributions:

- We propose a novel semantically secure AHE scheme called SAHE that supports addition as well as other homomorphic operations over encrypted data.
- We propose a novel semantically secure MHE scheme called SMHE that supports multiplication as well as other homomorphic operations over encrypted data.
- We introduce a set of compaction techniques to limit the overhead of ciphertext size for our proposed schemes.
- We show the design and evaluation of a system called **Symmetria** that uses our proposed schemes and employs a set of query optimizations to execute queries over encrypted data with little overhead.

In the rest of the paper, we first present background information and related work in §2 and then describe the design of SAHE and SMHE in §3. In §4 we provide an overview of Symmetria and in §5 we discuss implementation details of Symmetria. In §6 we empirically evaluate Symmetria and our proposed schemes. In §7 we conclude with final remarks.

2. BACKGROUND AND RELATED WORK

In this section, we first present pertinent background information on partially homomorphic encryption (PHE) and property-preserving encryption (PPE) schemes, and then overview most closely related existing PHE-based systems.

2.1 PHE and PPE

PHE [37] schemes have the property that their ciphertexts can be altered in certain ways so that the underlying plaintext is also changed in predictable and controllable ways. An encryption scheme is said to be partially homomorphic with respect to certain operations if it enables those operations on encrypted data by altering a given ciphertext or combining ciphertexts. E.g., if **enc** and **dec** denote encryption and decryption functions respectively, then an encryption scheme is said to be homomorphic with respect to addition if $\exists \psi$ s.t.

$$\operatorname{dec}(\operatorname{enc}(m_1) \ \psi \ \operatorname{enc}(m_2)) = m_1 + m_2$$

Such an encryption scheme is thus called an additive homomorphic encryption (AHE) scheme. Similarly, a scheme is said to be a multiplicative homomorphic encryption (MHE) scheme, if $\exists \chi$ s.t.

$$\mathsf{dec}(\mathsf{enc}(m_1) \ \chi \ \mathsf{enc}(m_2)) = m_1 \times m_2$$

Several PHE schemes support various homomorphic operations. The Paillier [33] cryptosystem is an AHE scheme, the security of which is based on the decisional composite residuosity assumption. Informally, the additive homomorphic property of Paillier can be described as:

$$\operatorname{dec}(\operatorname{enc}(m_1) \times \operatorname{enc}(m_2) \mod N^2) = m_1 + m_2 \mod N$$

where m_1 and m_2 are two plaintext values and N is the public key. Another useful homomorphic feature of Paillier is multiplication between a ciphertext and a plaintext value:

$$\mathsf{dec}(\mathsf{enc}(m_1)^{m_2} mod N^2) = m_1 imes m_2 mod N$$

Other AHE schemes include Benaloh [14] and Damgård-Jurik [11]. These schemes support addition and subtraction between ciphertexts, addition and multiplication between a ciphertext and a plaintext, and negation of ciphertexts.

The ElGamal [12] cryptosystem is an asymmetric MHE scheme based on the computational Diffie-Hellman assump-

¹https://github.com/ssavvides/symmetria.

tion, allowing homomorphic multiplications. In addition, raising an encrypted value to a plaintext value and decrypting the result gives the exponentiation of the first value raised to the power of the second one. RSA [38] (unpadded), similarly supports multiplication and division (multiplication with the inverse) between two ciphertexts, multiplication and exponentiation between a ciphertext and a plaintext, and computing of the multiplicative inverse of a ciphertext.

The AHE and MHE cryptosystems mentioned above are asymmetric and their security is based on mathematical problems that are hard to solve for large numbers. For example, if the public key N of Paillier can be factored, the security of the cryptosystem no longer holds. For small values of N, solving this problem is trivial and therefore sufficiently large numbers should be used. As of the time of this writing, NIST recommends N to be at least 2048-bits long [3], making the mod N^2 homomorphic computations in Paillier 4096-bits long. Furthermore, NIST recommends that 2048-bits keys should not be used beyond the year 2030, with other organizations suggesting that 3072-bit long keys are already needed today [7, 10]. Oftentimes, these cryptosystems and the associated homomorphic computations are implemented using an arbitrary-precision arithmetic library such as GMP, or the BigInteger arithmetic primitive in Java. These libraries, though highly optimized, can exhibit high overheads for computations involving large numbers.

ASHE [34] is a recent AHE scheme which is *symmetric*. ASHE was designed for summations and only supports additions between two ciphertexts. In this work, we present not only a symmetric AHE scheme but also a symmetric MHE scheme that supports homomorphic multiplication. But most importantly, unlike ASHE, SAHE was designed to support all homomorphic operations that previous (asymmetric) AHE allowed, and not only addition between two ciphertexts.

Another category of cryptosystems that allow computations over encrypted data is property-preserving encryption (PPE). Ciphertexts of PPE schemes preserve some properties of the underlying plaintext allowing operations to be applied on the ciphertext such as equality and order comparisons, and search over encrypted data. For example, deterministic schemes (DET) can be used to support equality comparisons since encrypting the same plaintext will always yield the same ciphertext. Boldyreva et al. [4, 5] present an order-preserving encryption (OPE) scheme that allows order comparisons over encrypted data. Lastly, the SWP [41] cryptosystem is an example of a searchable encryption (SRCH) scheme that allows searches over encrypted strings. In the past, PPE schemes have been criticized for having low-security guarantees [8, 18, 20, 31], but recent work has introduced new PPE schemes that provide semantic security [35].

2.2 PHE-based Systems

Several PHE-based systems have been proposed that show how PHE can be made practical for confidentiality-preserving computations. CryptDB [36] extends a MySQL database to enable the execution of SQL queries in a confidentialitypreserving manner by using PHE. CryptDB uses Paillier to perform homomorphic additions but does not support any MHE scheme so multiplications cannot be performed. Arx [35] is a PHE-based system that extends MongoDB to support confidentiality-preserving database applications. Arx introduces the ArxEq and ArxRange primitives that enable equality and range computations over data encrypted using semantically secure encryption. We plan to extend Symmetria to support these primitives and replace the PPE schemes currently used which offer lower security guarantees. Since the focus of Symmetria is to demonstrate the benefits of using our symmetric PHE schemes over asymmetric PHE schemes, in the current version of Symmetria we chose to use PPE schemes whose implementations are publicly available. Seabed [34] is a system built on Apache Spark that introduced ASHE. ASHE and Seabed were designed with the goal of performing large-scale summations efficiently and have limited homomorphic expressiveness, as opposed to Symmetria that uses SAHE and SMHE that support a wider range of homomorphic operations. Cuttlefish [39] is a PHE-based system built on Apache Spark. Cuttlefish uses ElGamal for multiplications and both Paillier and ASHE for additions. While ASHE is faster because it is symmetric, it does not support subtraction operations or multiplications between a ciphertext and a plaintext value. SAHE supports all these operations as well as others, while being symmetric.

Several techniques have been proposed to compact ciphertexts of PHE schemes [15, 21, 30]. Most prominently, Ge and Zdonik [15] describe a technique that allows packing multiple plaintext values into a single ciphertext, thereby amortizing the ciphertext size overhead. The technique applies to asymmetric AHE schemes but not to symmetric AHE schemes or to MHE schemes. In addition, it introduces the issue of overflows for aggregation functions and limits expressiveness, as homomorphic operations cannot be performed to packed values individually, nor can some homomorphic operations such as multiplication with plaintext, negation, or subtraction be performed on packed ciphertexts. Monomi [48] is a PHE-based database system using the above-mentioned packing technique to reduce ciphertext size overhead and accelerate summations. Similarly, Liu et al. [27] employ packing to reduce the overheads of computing the trajectory similarity function over encrypted data. These works inherit the drawbacks of the packing technique described above. In contrast, the compaction techniques we introduce herein (§ 3.4) do not limit the expressiveness of homomorphic operations, and do not suffer other drawbacks of packing multiple values into one ciphertext.

2.3 Trusted Hardware-based Systems

Recently, trusted execution environments (TEEs) like Intel Software Guard eXtensions (SGX) [29] have also gained traction for performing secure computations in the cloud. SGX allows the creation of isolated areas of execution, called enclaves. Within enclaves, operations can be performed on sensitive data in a way protected against accesses from the outside environment, including those from the operating system and hypervisor. Works like VC3 [40], Opaque [50], and TensorScone [24], leverage SGX for ensuring data confidentiality and integrity during computation in untrusted environments. In this work, we focus on a software-only solution based on PHE and PPE that does not require specialized hardware and place emphasis on improving the performance and expressiveness of existing PHE and PPE based systems.

3. SYMMETRIC PHE

We describe two novel *symmetric* PHE schemes that can replace existing asymmetric schemes such as Paillier and ElGamal used for arithmetic operations in state-of-the-art PHE-based systems [13, 36, 39, 47, 49]. Our schemes trade strict compactness for quantitatively good compactness in practice, which is supported by a set of compaction techniques. These techniques allow our schemes to achieve better performance than asymmetric schemes without sacrificing expressiveness, as they retain the full range of homomorphic operations of traditional PHE schemes.

3.1 Symmetric AHE

Symmetric additive homomorphic encryption (SAHE) is a symmetric AHE scheme that allows homomorphic additions.

3.1.1 Scheme description

SAHE is defined in the abelian additive group \mathbb{Z}_N of order N > 1. Let $F : \{0,1\}^n \times \{0,1\}^n \to \mathbb{Z}_N$ defined as $F_k(x) = F(k,x)$ be a pseudorandom function (PRF) mapping strings of length n to elements of \mathbb{Z}_N , where n is the security parameter which dictates the length of the key and must be set according to the application needs. Below we formally define the scheme $\Pi_{SAHE} = (\text{gen}, \text{enc}, \text{dec})$.

gen (1^n) : output uniform $k \in \{0,1\}^n$ as the symmetric key. enc(k,m): on input key $k \in \{0,1\}^n$ and message $m \in \mathbb{Z}_N$ choose uniform $r \in \{0,1\}^n$ and output the ciphertext

so se uniform
$$r \in \{0, 1\}^{\circ}$$
 and output the ciphert

$$c := \langle (m + F_k(r)) \mod N, [r], \varnothing \rangle$$

The resulting ciphertext c is a triplet $\langle v, l_p, l_n \rangle$ where v is the obfuscated value, l_p is a list of identifiers (ids) each of which is used to generate a random element in \mathbb{Z}_N and added to m, and, l_n is a list of ids used to generate a random element in \mathbb{Z}_N and subtracted from m. When initially encrypting a value m, there is only one id added to m to get $v = (m + F_k(r)) \mod N$ and therefore $l_p = [r]$ and $l_n = \emptyset$ (empty list).

 $\operatorname{dec}(k, c)$: on input key $k \in \{0, 1\}^n$ and ciphertext $c = \langle v, l_p, l_n \rangle$ output the plaintext message

$$m := (v - \sum_{r_1 \in l_p} F_k(r_1) + \sum_{r_2 \in l_n} F_k(r_2)) \mod N$$

3.1.2 Proof of security

We provide proof of the security of SAHE and show that it is secure under the assumption that F_k is a secure PRF. Note that lists l_p and l_n hold ids that act as inputs to F_k and can remain in the clear as long as the key k is kept secret.

THEOREM 1. If F is a PRF, the cryptographic construction Π_{SAHE} is a semantically secure under CPA (IND-CPA) private-key encryption scheme for messages of \mathbb{Z}_N .

PROOF. We assume a probabilistic polynomial-time (PPT) adversary A that attempts to break the semantic security of $\Pi_{SAHE} = (\text{gen}, \text{enc}, \text{dec})$, for the rest of the proof simply Π . We denote the probability of A succeeding as $Pr[CPA_{\Pi}^{A} = 1]$ where CPA_{Π}^{A} is the CPA indistinguishability experiment. We show that if A can break the semantic security of Π we can construct an attacker that can distinguish a PRF F from a truly random function f.

We proceed by a proof by reduction by using adversary A as a subroutine to construct a distinguisher D. D has oracle access to a function $\mathcal{R} : \{0,1\}^n \to \mathbb{Z}_N$ which can either be a PRF or a truly random function, and the goal of D is to determine which is the case. D emulates CPA^A and when A requests the encryption of a message $m \in \mathbb{Z}_N$, D generates a uniform string $r \in \{0,1\}^n$, uses the oracle to get $x := \mathcal{R}(r)$, and returns the ciphertext $\langle (m+x) \mod N, [r], \varnothing \rangle$ to A. Recall that n is chosen such that adversary A can

issue a polynomial number of such encryption requests which we denote as poly(n). This process is repeated until Aoutputs messages $m_0, m_1 \in \mathbb{Z}_N$. In return, D chooses a uniform bit $b \in \{0, 1\}$, a uniform string $r \in \{0, 1\}^n$, uses the oracle to get $x := \mathcal{R}(r)$, and returns the challenge ciphertext $\langle (m_b + x) \mod N, [r], \emptyset \rangle$ to A. Finally, when A outputs a bit b' indicating a guess on what message was encrypted, Doutputs 1 if b = b', and 0 otherwise.

Let Π be a cryptographic construction same as Π but where the PRF F is replaced with a truly random function f. Notice that if \mathcal{R} is a PRF then $x := F_k(r)$ and the emulation of CPA^A matches the experiment with construction Π , i.e., CPA^A_{Π} . Similarly, when \mathcal{R} is a truly random function then x := f(r) and the emulation of CPA^A matches the experiment with construction Π , i.e., CPA^A_{Π} . Therefore, by the assumption that F_k is a secure PRF, there exists negligible probability ϵ s.t.

$$|Pr[CPA_{\Pi}^{A}=1] - Pr[CPA_{\widetilde{\Pi}}^{A}=1]| \leq \epsilon$$
(1)

In the case of \mathcal{R} being a truly random function, we consider the probability of the adversary A having received a ciphertext encrypted using the same random string $r \in \{0,1\}^n$ as the one used to generate the challenge ciphertext $c = \langle (m + f(r)) \mod N, [r], \emptyset \rangle$. Note that in such event, A can infer $f(r) = (c - m) \mod N$ and use that to determine whether the challenge cipher was generated from m_1 or m_2 , winning the indistinguishability game. Since A can perform poly(n) number of encryption requests, and each encryption requires one out of a total of 2^n possible strings, such an event can occur with probability $\frac{poly(n)}{2^n}$. Therefore we have

$$Pr[CPA_{\widetilde{\Pi}}^{A}=1] \leq \frac{1}{2} + \frac{poly(n)}{2^{n}}.$$
(2)

Using Eq. 1 and Eq. 2 we get

$$Pr[CPA_{\Pi}^{A} = 1] \le \frac{1}{2} + \frac{poly(n)}{2^{n}} + \epsilon$$

where $\frac{poly(n)}{2^n} + \epsilon$ is negligible and therefore the advantage of adversary A is negligible. \Box

3.1.3 Homomorphic operations

Next, we describe the set of homomorphic operations $\Phi_{SAHE} = (\text{add}, \text{ adp}, \text{ mlp}, \text{ neg}, \text{ sub})$ of SAHE. In what follows, we use c to denote a ciphertext $(c = \langle v, l_p, l_n \rangle)$, c_1 and c_2 to denote a pair of ciphertexts $(c_1 = \langle v_1, l_{p_1}, l_{n_1} \rangle$ and $c_2 = \langle v_2, l_{p_2}, l_{n_2} \rangle$ and m to denote a plaintext value. As previously mentioned, ciphertexts are randomized and none of the operations in Φ_{SAHE} require the use of trapdoors.

add: This operation homomorphically adds two ciphertexts: $\operatorname{add}(c_1, c_2) := \langle (v_1 + v_2) \mod N, l_{p_1} \cup l_{p_2}, l_{n_1} \cup l_{n_2} \rangle$

Here \cup indicates list concatenation.

adp: The <u>add</u> <u>plaintext</u> operation expresses a homomorphic addition between a ciphertext and a plaintext value:

$$adp(c,m) := \langle (v+m) \mod N, l_p, l_n \rangle$$

mlp: Even though SAHE is an AHE scheme, it supports multiplication between a ciphertext and a plaintext value through the mlp (<u>multiply plaintext</u>) operation:

 $\mathsf{mlp}(c,m) := \langle (v \times m) \mod N, l_1, l_2 \rangle$

where

$$l_1, l_2 = \begin{cases} l_n^{|m|}, l_p^{|m|} & \text{if } m < 0\\ \varnothing, \varnothing & \text{if } m = 0\\ l_p^{|m|}, l_n^{|m|} & \text{if } m > 0 \end{cases}$$
(3)

In the above, |m| denotes the absolute value of m and the list operation $l^{|m|}$ denotes |m-1|-fold concatenation of list l with itself. For example, if $l = [r_1, r_2]$ and m = 3 then $l^{|m|} = [r_1, r_2, r_1, r_2, r_1, r_2]$. This example shows that the lists can grow quickly as more homomorphic operations are performed. In § 3.4 we introduce a set of compaction techniques to alleviate this concern.

neg: We define **neg**, the unary operation of negating an encrypted value based on the **mlp** function:

$$eg(c) := mlp(c, -1) = \langle -v \mod N, l_n, l_p \rangle$$

sub: We define the subtraction operation sub based on the homomorphic operations add and neg:

$$\begin{aligned} \mathsf{sub}(c_1, c_2) &:= \mathsf{add}(c_1, \mathsf{neg}(c_2)) \\ &= \langle (v_1 - v_2) \bmod N, l_{p_1} \cup l_{n_2}, l_{n_1} \cup l_{p_2} \rangle \end{aligned}$$

3.2 Symmetric MHE

Symmetric multiplicative homomorphic encryption (SMHE) is a symmetric MHE scheme that allows homomorphic operations with respect to multiplication.

3.2.1 Scheme description

SMHE is defined in the abelian multiplicative group \mathbb{Z}_N^* of order N > 1. Just like with SAHE, we assume there exists a PRF $F_k(x) = F(k, x)$ generating elements of \mathbb{Z}_N^* , and use a fixed generator g for the selected group order. Below we formally define the scheme $\Pi_{SMHE} = (\text{gen}, \text{enc}, \text{dec})$.

 $gen(1^n)$: same as SAHE gen function.

enc(k, m): on input key $k \in \{0, 1\}^n$ and message $m \in \mathbb{Z}_N^*$ choose uniform $r \in \{0, 1\}^n$ and output the ciphertext

$$c:=\langle (m imes g^{F_k(r)}) mod N, [r], arnothing
angle$$

Ciphertext c is a triplet $\langle v, l_p, l_n \rangle$ but for SMHE l_p denotes the list of ids each of which is used to generate a random element in \mathbb{Z}_N^* which is then raised to the power of g and multiplied by m to generate the obfuscated value v. l_n is similar except that the generated value $g^{F_k(r)}$ for a given r is inverted before being multiplied by m.

 $\operatorname{\mathsf{dec}}(k,c)$: on input key $k \in \{0,1\}^n$ and ciphertext $c = \langle v, l_p, l_n \rangle$ output the plaintext message

$$m:=(v\times\prod_{r_1\in l_p}g^{-F_k(r_1)}\times\prod_{r_2\in l_n}g^{F_k(r_2)}) \bmod N$$

3.2.2 Proof of security

Here, we prove that SMHE is secure under the assumption that F is a secure PRF.

THEOREM 2. If F is a PRF, the cryptographic construction Π_{SMHE} is a semantically secure under CPA (IND-CPA) private-key encryption scheme for messages of \mathbb{Z}_N^* .

PROOF. As the proof is similar to the one for SAHE, we only provide the parts specific to SMHE. First, we note that since g is a generator of the group \mathbb{Z}_N^* every element in the group can be expressed via g^x for some x. Thus, assuming a truly random function f, for uniform r, $g^{f(r)}$ is uniformly distributed in the group. Since g is a generator and $m \in \mathbb{Z}_N^*$ then $m = g^x$ for some x. Therefore $m \times g^{f(r)} = g^x \times g^{f(r)} =$ $g^{x+f(r)} \mod N$ is uniform as well. The rest of the proof proceeds in three steps like the proof for SAHE by 1. proving that replacing f with the PRF F_k only gives the adversary negligible advantage, 2. showing that a modified Π_{SMHE} scheme where $F_k(r)$ is replaced with f is semantically secure and 3. showing that poly(n) is the upper bound on the number of queries that the adversary can send to the oracle of the indistinguishability game, and the only way the adversary has an advantage is when the same r is used more than once which happens only with negligible probability. \Box

3.2.3 Homomorphic operations

In this section, we describe the set of homomorphic operations $\Phi_{SMHE} = (\text{mul, mlp, pow, inv, div})$ of SMHE.

nul: This operation denotes homomorphic multiplication:

$$\operatorname{mul}(c_1, c_2) := \langle (v_1 \times v_2) \mod N, l_{p_1} \cup l_{p_2}, l_{n_1} \cup l_{n_2} \rangle$$

mlp: The <u>multiply plaintext</u> operation denotes homomorphic multiplication between a ciphertext and a plaintext:

$$\mathsf{mlp}(c,m) := \langle (v \times m) \mod N, l_p, l_n \rangle$$

pow: SMHE supports exponentiation between a ciphertext and a plaintext value through the **pow** operation:

$$\mathsf{pow}(c,m) := \langle (v^m) \mathsf{ mod } N, l_1, l_2 \rangle$$

Here l_1 and l_2 are computed as per Eq. 3.

inv: The multiplicative inverse of a ciphertext can be homomorphically computed as follows:

$$\operatorname{inv}(c) := \operatorname{pow}(c, -1)$$

div: Finally, homomorphic division (multiplication with the inverse) is defined as:

$$\operatorname{div}(c_1,c_2) := \operatorname{mul}(c_1,\operatorname{inv}(c_2))$$

3.3 Security Properties

We discuss some security properties of SAHE and SMHE.

3.3.1 Statefulness

So far the description of SAHE and SMHE required that we generate a uniform id $r \in \{0,1\}^l$ when encrypting, although the homomorphic operations and the security of our schemes do not require r to be uniform. It is sufficient for r to be unique across all values encrypted under the same key k. We can thus change the enc algorithm of both of our schemes from being randomized to being stateful. By keeping track of the last id used (for a given key k), ids can be chosen in an incremental fashion. By default, Symmetria uses the stateful implementation of enc for both SAHE and SMHE which leads to more regular lists of ids and make the application of the compactness techniques described in § 3.4 more effective.

3.3.2 Malleability and null values

Similar to other PHE schemes, homomorphic operations of our schemes are a result of the schemes being malleable and, therefore, allow some limited alterations to ciphertexts which result in predictable changes to the underlying plaintexts. As a result, stronger notions of security such as IND-CCA2 are not satisfied by our (or any previous) homomorphic schemes. Asymmetric schemes allow the creation of any ciphertext without access to the private key since encryption only requires the public key. So far the description of our schemes permits the creation of arbitrary ciphertexts without having access to the secret key k, despite our schemes being symmetric. For example, for either SAHE or SMHE and for any message $0 \leq m < N$, one can construct the ciphertext $c^* = \langle m, \emptyset, \emptyset \rangle$ where the obfuscated value of the ciphertext is set to m itself and both id lists l_p and l_n are empty. As per discussion so far, c^* is a legitimate ciphertext that once decrypted will return the message m, i.e., dec $(c^*, k) = m$. Ciphertexts with both id lists being empty can also occur by applying operations on a non-empty ciphertext. For instance, given a SAHE ciphertext c one can perform the operation $c_0 = mlp(c, 0)$ to get a valid (per discussion so far) ciphertext c_0 of the value 0. Then, by repeated application of $adp(c_0, 1)$ one can create a valid ciphertext for any value desired. To address this issue we treat ciphertexts with both id lists being empty in a special way. Given a ciphertext triplet $\langle v, l_p, l_n \rangle$, if both lists are empty ($l_p = \emptyset$ and $l_n = \emptyset$) the ciphertext is considered as a null ciphertext regardless of the value v. We denote the null ciphertext of SAHE and SMHE by \emptyset^A and \emptyset^M respectively. Decryption considers such ciphertexts as a special case. For both SAHE $dec(\emptyset^A, k) = 0$ (identity element of \mathbb{Z}_N) and SMHE $dec(\emptyset^M, k) = 1$ (identity element of \mathbb{Z}_N^*), the obfuscated value v is ignored.

3.4 Compaction Techniques

Ciphertexts of SAHE and SMHE are made up of three elements $\langle v, l_p, l_n \rangle$. The size of v depends on the choice of N and is fixed. The size of id lists l_p and l_n however can vary as homomorphic operations are performed. As l_p and l_n increase in size, the memory footprint of Symmetria grows. Moreover, the performance of homomorphic operations and decryption of ciphertexts degrades as these lists grow. To reduce these overheads we propose a set of techniques to compact these lists. We also define alternatives for enc and dec that leverage the techniques for improved performance.

3.4.1 List aggregation

Ids in lists l_p and l_n are used to generate a random value using a PRF F_k which in turn is used to apply an operation during encryption or decryption. For both SAHE and SMHE, lists l_p and l_n describe inverse operations. For instance, in SAHE, list l_p holds ids used to generate a random value *added* to the message m, and l_n holds ids used to generate a random value *subtracted* from m. These operations can cancel each other out, meaning that ids that appear in both lists can be safely removed. For example, the two lists $l_p = [r_1, r_2, r_3]$ and $l_n = [r_1, r_4]$ can be reduced to $l_p = [r_2, r_3]$ and $l_n = [r_4]$.

3.4.2 Id grouping

 $r_1 \in l_p$

Homomorphic operations of both SAHE and SMHE can lead to ids appearing multiple times in the same id list. For example, after an mlp operation using the SAHE scheme, each id will by copied multiple times in both id lists l_n and l_n , as per Eq. 3. To keep id lists more concise, same ids within a list can be grouped. For example, the list of ids $[r_1, r_1, r_1, r_2]$ can be represented as $[3:r_1,r_2]$. Here "3:" indicates the cardinality of the id r_1 . Furthermore, subsequent additions of the id r_1 to the list do not change the list size or the overall ciphertext size and instead, the cardinality of the id r_1 is increased. This technique not only reduces ciphertext footprint but also improves decryption performance as we describe next. We define a function $\mathcal{C}: \{0,1\}^l \to \mathbb{N}$ which, given an id $r \in \{0,1\}^l$, returns the cardinality of that id in the list, i.e., how many times that id appears in the list. Using this function, we can change the definitions of decryption functions (dec) for SAHE and SMHE respectively as follows:

$$\begin{split} m &:= (v - \sum_{r_1 \in l_p} \mathcal{C}(r_1) \times F_k(r_1) + \sum_{r_2 \in l_n} \mathcal{C}(r_2) \times F_k(r_2)) \bmod N \\ m &:= (v \times \prod g^{-\mathcal{C}(r_1) \times F_k(r_1)} \times \prod g^{\mathcal{C}(r_2) \times F_k(r_2)}) \bmod N \end{split}$$

 $r_2 \in l_n$

With this technique enabled, list concatenation (Eq. 3) can be efficiently implemented simply by updating id cardinalities.

3.4.3 Range folding

Sequences of consecutive ids is another pattern appearing in id lists, especially when ids are selected sequentially at encryption (see § 3.3.1). These sequences can be folded into a range of ids. E.g., consider the id list [2, 3, 4, 5, 8]. This list can be compacted to [2 - 5, 8]. In this example, subsequent additions of the ids 1 or 6 do not increase the size of the list as these ids will simply extend the existing range. To make this technique more effective, we generalize range folding to fold sequences of ids of any step and not just consecutive sequences, i.e., sequences of step 1. We have observed that in practice such patterns appear frequently, for instance when summing over a column, even after some rows are filtered.

3.4.4 Telescoping

To enable efficient execution of aggregation functions we use the telescoping technique. Consider the case where a sizable number of ciphertexts need to be summed. The ids of these ciphertexts can be consecutive in which case the resulting ciphertext will be compact due to the range folding technique discussed above. Even if this is the case, while decrypting, $F_k(r)$ needs to be generated for each id r even if r is folded in a range, making decryption impractical. To avoid this, we change the encryption functions for SAHE and SMHE respectively as follows:

$$\begin{aligned} c &:= \langle (m + F_k(r) - F_k(r+1)) \mod N, [r], [r+1] \rangle \\ c &:= \langle (m \times g^{F_k(r)} \times g^{-F_k(r+1)}) \mod N, [r], [r+1] \rangle \end{aligned}$$

Here, for each id r we generate $F_k(r)$ and $F_k(r+1)$ and apply inverse operations based on these, storing r in l_p and r+1 in l_n . For consecutive values of r, these operations cancel each other out based on our list aggregation technique, making decryption practical, at the expense of having to generate an additional pseudorandom number (PRN) at encryption.

3.4.5 Integer list compression

Our schemes can also benefit from compaction techniques proposed in previous work. In particular, the problem of efficiently compressing arrays of integers has been studied extensively [25, 26]. A requirement of such algorithms is that the integers of the array are not random. Our lists of ids satisfy this requirement by storing ids in non-decreasing order. Another requirement for these algorithms to be effective is that most integers in the array are small or gaps between them are small. These requirements are met by our schemes by choosing ids in an incremental manner (§ 3.3.1).

3.5 Scheme Compactness

As SAHE and SMHE are not strictly compact each application of a homomorphic operation might yield a ciphertext with increased size. Here we discuss how the compactness techniques proposed above can be applied to achieve homomorphic operations that are (quantitatively) compact in practice. Tab. 1 outlines the conditions for respective operations to retain compactness. Any homomorphic operation is compact if it involves (a) adding an id to the l_p list and the same id exists in the l_n list or vice-versa (§ 3.4.1); (b) adding an id to a list that already contains the id (§ 3.4.2); (c) adding an id to a list that falls within an existing range (§ 3.4.3).

 Table 1: Scheme operation compactness.

SAHE	SMHE	Compactness condition	
add	mul	Techniques $\S 3.4.1$, $\S 3.4.2$, and $\S 3.4.3$	
adp	mlp	By definition	
mlp	pow	Technique § 3.4.2	
neg	inv	By definition	
sub	div	Techniques $\S 3.4.1$, $\S 3.4.2$, and $\S 3.4.3$	

Some homomorphic operations of our schemes are also compact by definition. In particular, operations involving a single ciphertext (e.g., neg, inv) as well as operations involving a ciphertext and a plaintext value (e.g. SAHE operations adp and mlp, SMHE operations mlp and pow) are always compact. We evaluate the effects of (a) scheme compactness in terms of ciphertext size overhead and execution time in § 6.4 and (b) individual compaction techniques in § 6.6.

4. Symmetria: SYSTEM DESIGN

We first explain the threat model underlying our Symmetria system and then give an overview of its design, followed by details on how it transforms queries to operate over encrypted data using our schemes. Lastly, we introduce query optimizations based on the properties of our schemes applied by Symmetria at query transformation to improve performance.

4.1 Symmetria Threat Model

Symmetria aims to preserve data confidentiality in the presence of a semi-honest adversary. We assume the adversary has access to all cloud nodes and can observe data and query executions, including any intermediate results. In addition, the adversary can observe data in transit between the cloud nodes and between the cloud nodes and the trusted application driver (described below). We assume the adversary cannot make changes to the queries, results, or data stored in the cloud, and consider integrity and availability attacks as well as access patterns and side-channel attacks out of scope. These attacks are orthogonal to our work and have been studied extensively in previous works. We plan to extend Symmetria to handle such attacks as part of our future work, using ideas from existing works [32, 40, 42, 43, 44, 50].

Symmetria uses a set of PPE schemes proposed in previous work for comparisons over encrypted data. Some of these schemes, e.g., OPE and DET, have been shown to provide lower security guarantees and even reveal partial plaintext [8, 18, 20, 31]. We plan to replace these schemes with more recent ones that offer semantic security [35] when their implementations become public.

Another concern specific to our encryption schemes SAHE and SMHE is the information leakage due to lack of strict compactness. Specifically, an adversary that observes the result of a query encrypted under SAHE or SMHE can infer the number of rows used to generate the result, e.g., how many items were added together to generate an aggregated sum. We note that an adversary that can observe the execution of queries, as well as any intermediate results (such as the adversary assumed in this work and by many previous PHEbased systems [36, 45, 48]), can infer the above information even when the homomorphic encryption schemes used are compact. But for a weaker adversary that can only observe the final results of the queries, the ciphertext size of our schemes can reveal the number of rows used to generate the final result. Under this weaker adversarial model, compact homomorphic schemes such as Paillier and ElGamal hide this information. Note that in some cases, our compaction techniques can hide the exact number of items used to generate a ciphertext, such as through the application of list aggregation (§ 3.4.1) and telescoping (§ 3.4.4).

4.2 Symmetria Design Overview

Fig. 1 shows the high-level design of Symmetria which includes several components divided between the trusted client and the untrusted cloud:

- Transformation module: transforms a query designed to operate over plaintext data to a query that operates on encrypted data to preserve data confidentiality.
- Encryption module: holds the cryptographic keys, encrypts traffic sent to the cloud and decrypts traffic received from the cloud. Also stores the crypto schema which contains the available encryptions of each column in the database.
- Application driver: deploys user-submitted queries to the cloud after they have been transformed. Receives encrypted results from the cloud, decrypts them using the encryption module and returns the results to the user.
- Compute service: a distributed computing framework that executes the submitted queries in the untrusted cloud. Utilizes a set of cryptographic user-defined functions (UDFs) to perform operations over encrypted data.

Encrypted database: holds the encrypted data.

Symmetria operates in two phases. During the setup phase when Symmetria is deployed, the data provider submits the plaintext data along with associated schema to the trusted client. The data provider also identifies operations expected to be performed on each column, following the approach used in previous work [19]. Using this information, the encryption module (1) encrypts each column under one or more encryption schemes according to the operations expected for it, (2) replaces the original column name with a random string for anonymization, and (3) uploads the encrypted data to the cloud. The encryption module also generates and stores a crypto schema with a mapping from each of the original column names to the resulting anonymized ones, along with the encryption scheme used to encrypt the column.

The second phase is the query execution phase. Users of Symmetria are oblivious of the underlying encryption strategy used to preserve data confidentiality and do not know the structure of the encrypted database. Users, therefore, submit queries designed to operate over plaintext data and assume that the database structure follows the original schema as submitted by the data provider during setup. When a query is submitted, the transformation module intercepts and transforms it into one that operates over encrypted data. To transform queries Symmetria follows the approach of previous works [22, 45, 46]. Firstly, the query is parsed into a logical plan which has the form of a directed acyclic graph (DAG) with nodes representing operations, column names, or literal values, and edges representing the flow of data in the query. For each operation in the logical plan, the transformation module considers the available schemes the involved column(s) are encrypted under. A column can either be a data column that exists in the encrypted database, in which case the available encryption schemes are kept in the crypto schema, or the column is derived from previous operations, in which case the lineage of the logical plan is used to derive the available encryption schemes of the col-



Figure 1: Symmetria system architecture. Dashed arrows indicate setup phase. Solid arrows indicate query execution phase.

umn. If one of the available encryption schemes supports the operation currently considered, then (i) the operation in the logical plan is replaced by the equivalent homomorphic operation, (ii) the involved column names are replaced with the (anonymized) names of the appropriately encrypted columns, and (iii) literal values are encrypted using the same scheme that supports the operation. If none of the available encryption schemes support the operation, the query is split (split execution [48]) and all remaining computation in the query is performed on the trusted client, after intermediate results are decrypted.

Once query transformation completes, the transformed query is passed to the application driver which deploys it in the cloud. The compute service in the cloud is an unmodified distributed computing framework able to perform homomorphic operations on encrypted data by means of cryptographic UDFs. These UDFs do not contain any sensitive information and are submitted to the compute service as part of the deployed application. Once the query execution completes, the (intermediate) encrypted results are sent back to the trusted client, decrypted, any remaining computation is performed, and the plaintext results are returned to the user.

4.3 Query Optimizations

We describe a set of optimizations, specific to SAHE and SMHE, applied to queries by the transformation module.

4.3.1 Expression re-writing

To improve performance for expressions involving multiple SAHE or SMHE homomorphic operations we introduce a set of re-writing rules (see Tab. 2 for SAHE and Tab. 3 for SMHE). The null operations category shows rules that replace expressions with null values during query transformation. Identity operation rules replace expressions that have no effect on the input ciphertext with the ciphertext itself. The operation replacing category replaces one expression with another equivalent expression with the goal of generating more uniform expressions that can be further optimized. Addition re-ordering and multiplication re-ordering rules re-arrange expressions to enable the application of other rewriting rules. Constant folding and constant factoring rules simplify operations involving constants. Term factoring rules replace an add operation with an mlp operation or remove an add operation altogether and similarly, term simplification rules remove or replace a **mul** operation with a **pow** operation.

Our re-writing rules go against traditional compilation. As, in general, plaintext multiplication is more expensive than addition [23], compilers replace integer multiplication with constants with consecutive additions or shift operations. E.g., $2 \times x$ is replaced by x + x (verified using gcc v7.3.0, Ubuntu v18.04). In contrast, in SAHE mlp is preferred over add as can be seen in rule category term factoring (Tab. 2);

Table 2: Re-writing rules for SAHE.	indicates that order
of operands is not relevant.	

Category	Before	After
Null operations	$\begin{array}{l} \operatorname{add}(\varnothing^A, \varnothing^A), \operatorname{adp}(\varnothing^A, m), \\ \operatorname{mlp}(\varnothing^A, m), \ \operatorname{mlp}(c, 0) \end{array}$	\varnothing^A
Identity operations	$\operatorname{add}(c, {arnothing}^A)^\dagger, \operatorname{adp}(c, 0), \\ \operatorname{mlp}(c, 1)$	с
Operation replacing	$\frac{neg(c)}{sub(c_1,c_2)}$	$\frac{mlp(c,-1)}{add(c_1,neg(c_2))}$
Addition re-ordering	$add(add(c_1,c_2),c_1)^\dagger \ add(adp(c_1,m),c_2)^\dagger$	$\begin{array}{l} add(add(c_1,c_1),c_2)\\ adp(add(c_1,c_2),m) \end{array}$
Constant folding	$adp(adp(c,m_1),m_2) \ mlp(mlp(c,m_1),m_2)$	$\frac{adp(c, m_1 + m_2)}{mlp(c, m_1 \times m_2)}$
Constant factoring	$add(mlp(c_1,m),mlp(c_2,m))$	$mlp(add(c_1,c_2),m)$
Term factoring	$\begin{array}{l} add(c,c) \\ add(mlp(c,m),c)^\dagger \\ add(mlp(c,m_1),mlp(c,m_2)) \end{array}$	$ \begin{split} & mlp(c,2) \\ & mlp(c,m+1) \\ & mlp(c,m_1+m_2) \end{split} $

similarly for SMHE pow is preferred over mul as shown in the rule category term simplification (Tab. 3).

4.3.2 Operation pipelining

A query can include expressions involving multiple homomorphic operations. For each operation on ciphertexts of the form $c = \langle v, l_p, l_n \rangle$, a modulo arithmetic computation is performed on v and, if the operation involves two ciphertexts, the two sets of id lists need to be merged, creating larger lists. Furthermore, any associated data-structures needed to enable compaction techniques (§ 3.4), e.g., data-structures used to hold cardinalities of ids, also need to be initialized and populated to create the resulting ciphertext. Finally, depending on the use case, the resulting ciphertext for each operation in the expression might have to be serialized.

To avoid repeated memory allocation and the creation of objects that hold intermediate results **Symmetria** pipelines all homomorphic operations in expressions, thereby amortizing their cost. All ciphertexts in an expression are collected and all homomorphic operations are applied in one go so that only data-structures for final ciphertexts need to be created. E.g., in the expression $c = \operatorname{add}(\operatorname{add}(\operatorname{add}(c_1, c_2), c_3), c_4)$, instead of creating intermediate ciphertexts for each add operation, all three operations are pipelined. Enough memory is pessimistically (assuming no compaction techniques apply) allocated to hold the data of all 4 input ciphertexts c_1-c_4 , and only the final ciphertext object c is created and serialized.

Operation pipelining is particularly effective in aggregation functions. Instead of deserializing and aggregating a single

Table 3: Re-writing rules for SMHE. † indicates that order of operands is not relevant.

Category	Before	After
Null operations	$\begin{array}{l} mul(\varnothing^M, \varnothing^M), mlp(\varnothing^M, m), \\ pow(\varnothing^M, m), pow(c, 0) \end{array}$	\varnothing^M
Identity operations	$mul(c, \varnothing^M)^\dagger, mlp(c, 1),$ pow(c, 1)	с
Operation replacing		pow(c,-1) $mul(c_1,inv(c_2))$
	$mul(mul(c_1,c_2),c_1)^\dagger \ mul(mlp(c_1,m),c_2)^\dagger$	$\frac{mul(mul(c_1, c_1), c_2)}{mlp(mul(c_1, c_2), m)}$
	$mlp(mlp(c,m_1),m_2)$ $pow(pow(c,m_1),m_2)$	$\frac{mlp(c, m_1 \times m_2)}{pow(c, m_1 \times m_2)}$
Term simplificat.	$\begin{array}{l} mul(c,c)\\ mul(pow(c,m),c)^{\dagger}\\ mul(pow(c,m_1),pow(c,m_2)) \end{array}$	pow(c, 2) pow(c, m + 1) $pow(c, m_1 + m_2)$

ciphertext value at a time into the intermediate aggregated result, a small number of serialized ciphertexts are kept in a cache. Once a threshold is reached, all are deserialized, aggregated together in one go as shown in the example above, and then added to the intermediate aggregation result.

4.3.3 Pre-computing PRNs

To encrypt a plaintext value or decrypt a ciphertext, both SAHE and SMHE require to generate one or more PRNs using $F_k(r)$ for a given r. When encryption is stateful (§ 3.3.1), r is chosen in an incremental fashion and is therefore known before a request to encrypt a value is made. In this case, to speed up encryption, **Symmetria** pre-computes and stores a small amount of such PRNs and uses them when a value needs to be encrypted. Similarly, decryption can be sped up by pre-computing $F_k(r)$ if the value r is known.

4.3.4 Plaintext multiplication

The mlp operation that allows multiplication between a ciphertext and a plaintext value is supported by both SAHE and SMHE. At query transformation, Symmetria chooses the most appropriate scheme for mlp depending on what other operations the corresponding column is involved in. This allows Symmetria to perform as much computation as possible over encrypted data in the cloud and prevents returning to the trusted client before strictly necessary. E.g., in the expression $2 \times (a + b)$ SAHE is used to perform the multiplication with the constant 2 as a+b involves an addition that is only supported by SAHE. On the other hand, in the expression $2 \times a \times b$ SMHE is used so the entire expression can be executed in the cloud. If there are no conflicting operations and either of the schemes can be used, to perform mlp, Symmetria uses SMHE as it is more performant (§ 6.3).

5. IMPLEMENTATION

In this section, we provide implementation details on our schemes SAHE and SMHE, as well as our Symmetria system.

5.1 Encryption Schemes

We implemented SAHE and SMHE and all associated compaction techniques in Java and using AES as PRF ${\cal F}_k.$

Each ciphertext triplet $\langle v, l_p, l_n \rangle$ includes an 8-byte integer value v and two id lists l_p and l_n which are implemented using immutable arrays. We use a separate hashmap to map ids to their cardinalities. To keep this map small, we assume a default cardinality of 1 and only keep track of higher cardinalities. To compress the list of ids we implemented a custom compression function that implements the compaction technique discussed in $\S 3.4.3$. We further compress the arrays of ids as described in §3.4.5 using the JavaFastP- FOR^2 library v0.1.12 and enable the compression codecs for differential coding, variable byte and fast PFOR (patched frame-of-reference). To speed up serialization/deserialization we implemented custom serialization functions for both SAHE and SMHE ciphertext objects. We also incorporate 3 PPE schemes, namely deterministic AES (ECB mode), OPE [4, 5], and SWP [41], as well as two asymmetric PHE schemes, namely Paillier [33] and ElGamal [12], which we use to compare against SAHE and SMHE. For Paillier and ElGamal we use 2048-bit long keys and use the Java BigInteger class for arbitrary-precision arithmetic computations.

5.2 Trusted Client

The trusted client contains the transformation module, encryption module, and application driver, and is therefore deployed in a trusted node. We implemented the transformation module by extending Apache Spark's Catalyst optimizer. By default, Catalyst applies a set of transformation rules to create a logical plan out of the query, optimize it, and turn it into an executable physical plan [2]. We created a set of transformation rules by extending the Spark Rule[LogicalPlan] class and use these rules to carry out the transformation of the query to one that operates over encrypted data. The query optimizations proposed in $\S4.3$ are also implemented as transformation rules. We register these rules with the Catalyst optimizer externally, i.e., without modifications to Spark, so that they are recursively applied to all nodes of the logical plan. We implement the application driver by extending the Spark driver. The application driver deploys the transformed query in the cloud and receives the encrypted results. The driver loads these results into a Scala parallel collection with default parallelization. The results are then decrypted in parallel and any remaining computation (see § 4.2) is performed in parallel before final results are returned.

5.3 Untrusted Cloud

The compute service runs an unmodified Apache Spark service. To enable the use of homomorphic operations of the schemes employed we created a Spark UDF function for each operation. For non-aggregation operations, we created a Spark UDF by extending the Spark UserDefinedFunction class; for aggregation functions such as summation and product we extend the UserDefinedAggregateFunction class. These functions are again externally registered.

6. EVALUATION

In this section, we evaluate the expressiveness and performance of our proposed schemes SAHE and SMHE as well as our prototype system **Symmetria**. We first assess the expressiveness of our schemes compared to previous schemes (§ 6.2). Then, we evaluate the performance of individual operations of our schemes (§ 6.3) and quantify the effect of

²https://github.com/lemire/JavaFastPFOR.

Table 4: Expressiveness comparison.Type indicateswhether a scheme is symmetric (sym) or asymmetric (asym).(a) AHE(b) MHE

(')			(
Paillier ASHE SAHE			1	ElGamal	SMHE	
Type	asym	sym	sym	Type	asym	sym
add	1	✓	✓	mul	1	✓
adp	✓	X	1	mlp	1	1
mlp	1	X	1	pow 🗸		1
neg	1	X	1	inv 🗸		1
sub	1	×	1	div	1	1
				-		

scheme compactness compared to other compact and noncompact schemes (§ 6.4). We then examine the encryption time and ciphertext size overhead of **Symmetria** compared to previous systems (§ 6.5) and finally we assess the overall efficiency of **Symmetria** and the performance benefits of our proposed compaction techniques and query optimizations by comparing its end-to-end latency with other systems (§ 6.6).

6.1 Evaluation Setup

We conduct all our experiments on Amazon EC2. We use a cluster of 10 m5.2xlarge (8 vCPUs, 32 GB RAM) slave nodes running Apache Spark v2.4.0 and deploy the Spark driver on a separate m5.4xlarge (16 vCPUs, 64 GB RAM) node. The Spark driver is assumed to be trusted and handles the decryption of the results. Decryption is parallelized and utilizes all available CPUs on the driver. We set up Spark to deploy a single executor per slave node with 8 tasks each (1 for each CPU) and a total of 28 GB memory. We load data into Hadoop HDFS (Hadoop v2.9.2) with replication factor 3 and use the Apache Parquet storage format to compress data. We carry out comparisons of primarily 3 system setups:

- **Plaintext** setup does not use any encryption and does not offer any confidentiality guarantees,
- Symmetria utilizes our proposed schemes SAHE and SMHE for arithmetic operations, and
- Asym utilizes the asymmetric schemes Paillier and ElGamal for arithmetic operations

We use two industry-adopted benchmarks for our evaluations. (1) TPC-H is a decision support benchmark comprising 22 queries. We use TPC-H v2.18 at a scale of 100 which uses over 100 GB of input data. (2) TPC-DS is a benchmark for big data decision solutions comprising 100 queries. Due to lack of space, we show the results of 19 of the TPC-DS queries as done in previous work [9]. We use TPC-DS v2.10 at a scale of 100 which uses 38.6 GB of plaintext input data.

Unless stated otherwise, reported execution times in our experiments are the average of 5 executions.

6.2 Scheme Expressiveness

Before conducting an empirical evaluation, we first show a simple comparison of our schemes with previous homomorphic schemes in terms of expressiveness. Tab. 4 shows the supported operations of various (a) AHE and (b) MHE schemes. As can be seen from the table, SAHE supports all homomorphic operations that Paillier supports though being symmetric. ASHE [34] is another AHE scheme that is symmetric like SAHE, but unlike SAHE, it has limited expressiveness and only supports additions between two ciphertexts (add). SMHE supports all homomorphic operations that El-Gamal supports though being symmetric. We know of no other symmetric MHE scheme to compare against SMHE.

6.3 **Operation Execution Times**

We now examine the execution times of individual operations in the sets Π_{SAHE} , Φ_{SAHE} , Π_{SMHE} , and Φ_{SMHE} . We compare these against the equivalent operations of the stateof-the-art asymmetric schemes Paillier and ElGamal. Paillier encryption and its associated homomorphic operations require arbitrary-precision arithmetic operations and modulo operations on the square of the public key. This means that the length of Paillier ciphertexts are 4096 bits long. ElGamal ciphertexts are made of two parts, each 2048 bits long. The increased ciphertext size and arbitrary-precision arithmetic operations required to achieve the homomorphic operations of Paillier and ElGamal lead to significant overhead. We also include a comparison against "Packed Paillier" which implements the packing technique introduced by Ge and Zdonik [15]. We pack 21 plaintext values in a ciphertext and include 32 bits of padding to account for overflows. Tab. 5 shows the results of this evaluation. This experiment was run on a single m5.2xlarge node. Each reported execution time is the average of 1 million executions after 100 warm-up executions. We observe that SAHE's encryption and decryption are 4 orders of magnitude faster than Paillier's. These can be made even faster by pre-computing the required PRNs as explained in $\S4.3.3$. Packed Paillier improves the performance of encryption and decryption significantly, but SAHE remains 2 orders of magnitude faster than Packed Paillier. Performing a homomorphic addition using Paillier requires a multiplication of the two ciphertexts modulo the square of the public key which is a 4096-bit value. This makes homomorphic addition of SAHE $76 \times$ faster than Paillier or $4 \times$ faster than Packed Paillier. The performance gains of other operations in Φ_{SAHE} compared to Paillier are even more dramatic because in Paillier these operations require exponentiation modulo the square of the public key which is a very expensive operation. We note that Packed Paillier does not support the operations mlp, neg, and sub. Due to this limitation, we do not use Packed Paillier for the rest of our evaluations. We observe similar results when comparing SMHE to ElGamal. Encryption and decryption functions of SMHE are 3 orders of magnitude faster than ElGamal. Arithmetic operations in Φ_{SMHE} range from $2 \times$ faster to $230 \times$ faster compared to the equivalent operations with ElGamal.

6.4 Effect of Non-Compactness

Next, we examine how the lack of compactness in our schemes affects ciphertext size and execution time and how effective our compaction techniques are in mitigating these effects. We perform a summation involving 1 million rows and to examine how our schemes behave when some of these rows are filtered out, we randomly sample the rows and perform a summation over the selected rows. We measure the size of the resulting ciphertext and the overall time taken to perform the summation as we change the sampling size from 5% to 100%. We compare Symmetria with all techniques enabled against Asym which uses asymmetric schemes that are compact. To examine the other extreme we also compare Symmetria against a Strawman setup which uses a naïve construction for arithmetic operations that encrypts individual values using the AES (CBC mode) block cipher and performs homomorphic operations by concatenating operators and encrypted operands. Unlike Symmetria, Strawman has no compaction techniques to limit ciphertext expansion. Fig. 2 shows the results of this evaluation. We observe that

Table 5: Operation execution times of SAHE and SMHE compared to asymmetric schemes. All reported times are given in nanoseconds followed by the relative standard error. Values in parentheses indicate pre-computation.

	J		1		1 1	
	Paillier	Packed Paillier	SAHE		ElGamal	SMHE
enc	$17285376 \pm 0.13\%$	$880921 \pm 0.11\%$	$1321~(63)~\pm~1.43\%$	enc	$8700278 \pm 0.04\%$	$2974~(752)~\pm~0.29\%$
dec	$16390295~\pm~0.01\%$	$781727~\pm~0.01\%$	$1202~(153)~\pm~4.18\%$	dec	$4768193 \pm 0.02\%$	$3090~(1420)~\pm~0.23\%$
add	$34807 \pm 1.37\%$	$1666 \pm 1.21\%$	$457 \pm 3.10\%$	mul	$25803~\pm~0.16\%$	$419 \pm 0.92\%$
adp	$917141 \pm 2.38\%$	$104775~\pm~0.95\%$	$71 \pm 0.37\%$	mlp	$678~\pm~1.17\%$	$371 \pm 0.11\%$
mlp	$857943 \pm 2.54\%$	_	$406 \pm 0.18\%$	pow	$505675~\pm~2.53\%$	$2856~\pm~0.37\%$
neg	$1370859~\pm~0.07\%$	_	$397 \pm 0.11\%$	inv	$809711 \pm 0.09\%$	$3529 \pm 0.24\%$
sub	$1408870~\pm~0.08\%$	_	$819 \pm 3.88\%$	div	$841260 \pm 0.14\%$	$4172 \pm 0.25\%$

Table 6: Encryption overheads. **Plaintext** (text) indicates uncompressed data. All other setups use Parquet to store compressed data. Time column refers to compression time for **Plaintext**, and adds encryption time for other setups.

Benchmark	System setup	Size	Time
	Plaintext (text)	$106.8~\mathrm{GB}$	_
TPC-H	Plaintext	$34.0~\mathrm{GB}$	$2.4 \min$
110-11	Asym	$363.7~\mathrm{GB}$	$84 \min$
	Symmetria	$67.8~\mathrm{GB}$	$14 \min$
	Plaintext (text)	38.6 GB	_
TPC-DS	Plaintext	$15.1~\mathrm{GB}$	$1.5 \min$
110-05	Asym	$482.4~\mathrm{GB}$	$228 \min$
	Symmetria	$39.7~\mathrm{GB}$	$4 \min$

the ciphertext size of the **Strawman** system setup increases linearly as the number of rows (sampling size) increases due to lack of any form of compaction. Even worse, its execution time increases exponentially since every subsequent addition operation occurs on a strictly larger aggregated result.

As expected, the ciphertext size of the Asym system setup remains constant (512 bytes) since it uses Paillier (with a 2048-bit modulo resulting in 4096-bit ciphertexts) to perform summation which is a compact scheme. Naturally, its execution time increases linearly with respect to sampling size, but due to the complex modulo operations involved in homomorphic addition the overall execution time is high.

Unlike Strawman, the compaction techniques of Symmetria keep the ciphertext size more bounded in practice, resulting in a much lower execution time. We also note that due to non-compactness, the ciphertext size of Symmetria is higher than that of Asym. At 50% sampling size, half of the rows are randomly filtered out making the range folding technique (3.4.3) less effective. Once the sampling size increases further, more sequences of consecutive ids occur, leading to more compact ciphertexts and therefore better execution times. At 100% selectivity, Symmetria achieves optimal compactness with the ciphertext becoming smaller than that of Asym (50 bytes compared to 512 bytes for Asym). Despite the overall higher ciphertext size of Symmetria, the execution time of Symmetria remains on average $30 \times$ faster than the Asym system. We further investigate the effect of increased ciphertext size in Symmetria and the associated network and decryption time overheads in $\S 6.6$.

6.5 Encryption Overhead

Next, we assess the time needed to encrypt the input data for the TPC-H and TPC-DS benchmarks and examine the size of the resulting encrypted data. We show the results of this evaluation in Tab. 6. TPC-H at scale 100 generates 106.8 GB of uncompressed data. We load this data into



Figure 2: Summation of 1 million rows as sampling size (x-axes) changes from 5% to 100%, with y-axes in log scale.

HDFS using the Parquet format which generates a total of 34 GB of compressed data and takes 2.4 min to complete the compression. We then encrypt the data using the two setups Asym and Symmetria. Under Asym, the size of the encrypted data is 363.7 GB which is over $5 \times$ larger than the encrypted data generated by Symmetria. In addition, Asym takes 84 min to complete encryption compared to 14 min with Symmetria. TPC-DS uses 38.6 GB of uncompressed data which, when compressed using Parquet, occupies 15.1 GB. Encrypting with Asym results in 482.4 GB of encrypted data which is a size overhead of $32 \times$ compared to plaintext data. In comparison, Symmetria generates only 39.7 GB of encrypted data with a size overhead of $2.6 \times$. Encryption time with Asym takes 228 min compared to only 4 min with Symmetria.

6.6 End-to-End Execution Latency

Finally, we use the TPC-H and TPC-DS benchmarks to compare end-to-end execution times of Plaintext, Asym, and Symmetria, by measuring the time from the point each query is submitted until the results are returned to the user after having been returned to the driver and decrypted. For this evaluation, Asym closely follows Monomi's choice of cryptosystems [48] which we further augment by an MHE scheme (ElGamal [12]) and a SRCH scheme (SWP [41]), implemented in Spark instead of the proposed PostgreSQL though. The resulting system called Monomi* uses split execution the same way as Symmetria. To further understand the performance benefits of each of our proposed compaction techniques and query optimizations, we repeat the experiment with different features enabled each time.

Fig. 3 shows the results of this experiment for TPC-H. To be able to evaluate homomorphic operations that operate on a ciphertext value and a plaintext value such as adp and mlp we choose 4 columns out of a total of 61 columns that do not seem to hold sensitive data, namely the columns *tax*, *discount*, *available-quantity*, and *supplier-quantity* and leave them in plaintext (also for Monomi*). The Symmetria bars show the execution times of Symmetria with all compaction



techniques and query optimizations enabled, normalized to the **Plaintext** execution time (**Plaintext** execution times in absolute are shown in parentheses under the query names (x - x)axis)). Each subsequent Symmetria bar shows the normalized execution time with one less feature enabled. Specifically, Symmetria-ER shows the normalized execution time when expression re-writing (§4.3.1) is disabled, Symmetria-OP shows the normalized execution time when in addition to expression re-writing, operation pipelining $(\S 4.3.2)$ is disabled, in Symmetria-PC PFOR compression $(\S 3.4.5)$ is disabled and finally, in Symmetria-RC range compression ($\S 3.4.3$) is disabled. We keep some compaction techniques permanently enabled — list aggregation ($\S 3.4.1$), id grouping ($\S 3.4.2$), and telescoping $(\S 3.4.4)$ — as without these techniques the id lists of ciphertexts become too large and many of the queries time out. The average execution time overheads compared to plaintext execution time of Symmetria, Symmetria-ER, Symmetria-OP, Symmetria-PC and Symmetria-RC are $5.35 \times$, $5.40 \times$, $9.50 \times$, $10.57 \times$, and $11.39 \times$ respectively. In comparison the average overhead of Monomi* is $20.39 \times$. We further observe that for Symmetria, the bulk of the work is done in the untrusted cloud. On average, only 1% of the overall query execution is spent on the trusted client side. This time includes communication overhead and decryption of final results. A notable exception is Q22 where 12.96% of the time is spent on the client side.

Fig. 4 shows the results of the same experiment on 19 queries of TPC-DS. TPC-DS uses 9 input tables and a total of 166 columns. A single column, *sales-price*, remains in plaintext. We show again the breakdown of Symmetria with some features disabled. Expression re-writing did not apply to any of the queries. The average execution time overheads compared to Plaintext execution time of Symmetria, Symmetria-PC and Symmetria-RC are $2.73\times$,

 $2.95\times$, $3.66\times$, and $4.45\times$ respectively and the average execution time overhead of **Monomi**^{*} is $19.16\times$. The average time spent on client side is only 2.12% for Symmetria.

Overall, Symmetria with all compaction techniques and query optimizations enabled is on average $3.8 \times$ faster on TPC-H queries than the state-of-the-art PHE-based systems using asymmetric schemes, and $7 \times$ faster on TPC-DS queries. This demonstrates the practicality of our schemes SAHE and SMHE despite the lack of strict ciphertext compactness and shows the effectiveness of our proposed optimizations.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we tackle the important problem of preserving the confidentiality of sensitive information while executing queries on it in an untrusted cloud. We introduce two novel symmetric PHE schemes that allow us to efficiently perform operations over encrypted data. Our schemes are faster compared to existing asymmetric PHE schemes and do not compromise homomorphic expressiveness. Our system Symmetria utilizes these schemes to enable the execution of queries over encrypted data. Symmetria achieves average speedups of 3.8× and 7× over state-of-the-art asymmetric PHE schemes on the standard TPC-H and TPC-DS benchmarks respectively. Besides possible adoption of other PPE schemes proposed recently [35], as part of our future work, we are investigating stronger security models and consider combining our proposed schemes with complementary techniques such as ORAM [32, 42, 44] to prevent active attacks.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This work was supported by NSF grant #1421910, ERC grant #617805, BMBF CRISP, and AWS Credits for Research.

9. **REFERENCES**

- A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In 6th Biennial Conference on Innovative Data Systems Research (CIDR'13), January 2013.
- [2] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In SIGMOD '15 Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pages 1383–1394, 2015.
- [3] E. Barker. Nist special publication 800–57 part 1, revision 4, 2016.
- [4] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In Int. Conf. on The Theory and Applications of Cryptographic Techniques (EUROCRYPT), pages 224–241, 2009.
- [5] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In Annual Int. Cryptology Conf. (CRYPTO), pages 578–595. Springer-Verlag, 2011.
- [6] D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. J. Wu. Private database queries using somewhat homomorphic encryption. In *International Conference* on Applied Cryptography and Network Security, pages 102–118. Springer, 2013.
- [7] BSI. Cryptographic methods: Recommendations and key lengths, bsi technical guideline, bsi tr-02102-1, 2019.
- [8] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In ACM Conference on Computer and Communications Security, pages 668–679. ACM, 2015.
- [9] Cloudera. A TPC-DS like benchmark for Cloudera Impala. https://github.com/cloudera/impala-tpcds-kit.
- [10] E. CSA. Algorithms, key size and protocols report, d5.4, 2018.
- [11] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *International* Workshop on Public Key Cryptography, pages 119–136. Springer, 2001.
- [12] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. Trans. on Information Theory, 31(4):469–472, 1985.
- [13] P. Eugster, S. Kumar, S. Savvides, and J. J. Stephen. Ensuring confidentiality in the cloud of things. *IEEE Pervasive Computing*, 18(1):10–18, 2019.
- [14] L. Fousse, P. Lafourcade, and M. Alnuaimi. Benaloh's dense probabilistic encryption revisited. In *AFRICACRYPT*, volume 6737 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2011.
- [15] T. Ge and S. Zdonik. Answering aggregation queries in a secure system model. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 519–530. VLDB Endowment, 2007.
- [16] C. Gentry. Fully homomorphic encryption using ideal lattices. In Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

- [17] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. *IACR Cryptology ePrint Archive*, 2012. Informal publication.
- [18] M. Giraud, A. Anzala-Yamajako, O. Bernard, and P. Lafourcade. Practical passive leakage-abuse attacks against symmetric searchable encryption. In 14th International Conference on Security and Cryptography SECRYPT 2017. SCITEPRESS-Science and Technology Publications, 2017.
- [19] Google. Encrypted bigquery client. https://github.com/google/encrypted-bigquery-client.
- [20] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *IEEE Symposium on Security and Privacy*, pages 655–672. IEEE Computer Society, 2017.
- [21] H. Hacigümüs, B. R. Iyer, and S. Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In Y. Lee, J. Li, K. Whang, and D. Lee, editors, *Database Systems for Advances Applications, 9th International Conference, DASFAA* 2004, Jeju Island, Korea, March 17-19, 2004, Proceedings, volume 2973 of Lecture Notes in Computer Science, pages 125–136. Springer, 2004.
- [22] M. Hauck, S. Savvides, P. Eugster, M. Mezini, and G. Salvaneschi. Securescala: Scala embedding of secure computations. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, pages 75–84, 2016.
- [23] Intel. Intel 64 and ia-32 architectures optimization reference manual. https://software.intel.com/enus/download/intel-64-and-ia-32-architecturesoptimization-reference-manual.
- [24] R. Kunkel, D. L. Quoc, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. Tensorscone: A secure tensorflow framework using intel sgx. arXiv preprint arXiv:1902.04413, 2019.
- [25] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice* and *Experience*, 45(1):1–29, 2015.
- [26] D. Lemire, N. Kurz, and C. Rupp. Stream vbyte: Faster byte-oriented integer compression. *CoRR*, abs/1709.08990, 2017.
- [27] A. Liu, K. Zheng, L. Li, G. Liu, L. Zhao, and X. Zhou. Efficient secure similarity computation on encrypted trajectory data. In J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, editors, 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015, pages 66–77. IEEE Computer Society, 2015.
- [28] P. Martins, L. Sousa, and A. Mariano. A survey on fully homomorphic encryption: An engineering perspective. *ACM Comput. Surv.*, 50(6):83:1–83:33, Dec. 2017.
- [29] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.
- [30] E. Mykletun and G. Tsudik. Aggregation queries in the database-as-a-service model. In E. Damiani and P. Liu, editors, *Data and Applications Security XX*, 20th

Annual IFIP WG 11.3 Working Conference on Data and Applications Security, Sophia Antipolis, France, July 31-August 2, 2006, Proceedings, volume 4127 of Lecture Notes in Computer Science, pages 89–103. Springer, 2006.

- [31] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 644–655, New York, NY, USA, 2015. ACM.
- [32] R. O. Oded Goldreich. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996.
- [33] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Int. Conf. on The Theory and Applications of Cryptographic Techniques (EUROCRYPT), pages 223–238, 1999.
- [34] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big data analytics over encrypted datasets with seabed. In Symp. on Op. Sys. Design and Implementation (OSDI), 2016.
- [35] R. Poddar, T. Boelter, and R. A. Popa. Arx: an encrypted database using semantically secure encryption. *PVLDB*, 12(11):1664–1678, 2019.
- [36] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: Processing queries on an encrypted database. *Commun. ACM*, 55(9):103–111, Sept. 2012.
- [37] R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. *Foundations* of Secure Computation, Academia Press, pages 169–179, 1978.
- [38] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, Feb. 1978.
- [39] S. Savvides, J. J. Stephen, M. S. Ardekani, V. Sundaram, and P. Eugster. Secure data types: A simple abstraction for confidentiality-preserving data analytics. In Symp. on Cloud Computing (SoCC), SoCC '17, pages 479–492, New York, NY, USA, 2017. ACM.
- [40] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis,

M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In Security and Privacy (SP), 2015 IEEE Symposium on, pages 38–54. IEEE, 2015.

- [41] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In Symp. on Security and Privacy (S&P), pages 44–55, 2000.
- [42] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In CCS '13 Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pages 247–258, 2013.
- [43] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious ram. arXiv preprint arXiv:1106.3652, 2011.
- [44] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: an extremely simple oblivious ram protocol. In CCS '13 Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pages 299–310, 2013.
- [45] J. J. Stephen, S. Savvides, R. Seidel, and P. Eugster. Practical confidentiality preserving big data analysis. In W. on Hot Topics in Cloud Computing (HotCloud), 2014.
- [46] J. J. Stephen, S. Savvides, R. Seidel, and P. T. Eugster. Program analysis for secure big data processing. In Int. Conf. on Automated Software Engineering (ASE), pages 277–288, 2014.
- [47] J. J. Stephen, S. Savvides, V. Sundaram, M. S. Ardekani, and P. Eugster. Styx: Stream processing with trustworthy cloud-based execution. In *Proceedings* of the Seventh ACM Symposium on Cloud Computing, SoCC '16, pages 348–360, New York, NY, USA, 2016. ACM.
- [48] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013.
- [49] D. Ulybyshev, A. O. Alsalem, B. Bhargava, S. Savvides, G. Mani, and L. B. Othmane. Secure data communication in autonomous v2x systems. In 2018 IEEE International Congress on Internet of Things (ICIOT), pages 156–163. IEEE, 2018.
- [50] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 283–298, 2017.