

Smart Crypt-Secure Storage and Sharing of Time Series Data Stream in Iot

¹Kadali Joshna Seshasai, ²S.Srinivas

^{1,2}Dept. of Computer Science & Engineering, KIET, Kakinada, AP, India

Abstract

A growing number of Industrial Internet of Things (IIoT) devices and services collect massive time-series data related to production, monitoring and maintenance. To provide ubiquitous access, scalability and sharing possibilities, the IIoT applications utilize the cloud to store collected data streams. However, secure storing of the massive and continuously generated data poses significant privacy risks, including data breaches for IIoT applications. Alongside, we need to protect the utility of the data streams by allowing benign services to access and run analytics securely and selectively.

To address this, we propose SmartCrypt, a data storing and sharing system that supports scalable analytics over the encrypted time-series data. SmartCrypt enables users to secure and fine-grain sharing of their encrypted data. Additionally, SmartCrypt guarantees data confidentiality in the presence of unauthorized parties by allowing end-to-end encryption using a novel symmetric homomorphic encryption scheme. We perform extensive experiments on a real-world dataset primarily to assess the feasibility of SmartCrypt for secure storing and sharing of IIoT data streams. The results show that SmartCrypt reduces query time by 17%, reduces range query time by 32%, improves throughput by 9% and scalability by 20% over the best performed scheme in the state-of-the-art.

Keywords

Access Control, Data Security, Homomorphic Encryption, Industrial Iot, Secure Sharing, Time-Series Data

1. Introduction

The growing adoption of Industrial Internet of Things (IIoT) in the smart manufacturing domain has spawned various emerging applications that collect sensitive time-series data. In smart manufacturing, time-series data are manufacturing data acquired by IIoT devices during systematic monitoring that capture the dynamic changes in production line over time. It is estimated that the total number of connected Internet of Things (IoT) devices will reach 83 billion by 2024 and over 70% of them will be IIoT devices [1]. Further, it is estimated that all these connected IoT devices will produce total data volume of 79.4 zettabytes [2]. The IIoT applications are suffering from critical data protection and data privacy risks [3]. For example, the Hot Strip Mill Process (HSMP) [4] involves several levels of product processing from raw materials to the final products, including process control, real-time control, manufacturing execution and business management level. All these various levels of production process generate sensitive time-series data. Most importantly, every data generated in the various levels of production process are new data instead of an update for already stored data. To estimate the volume of time-series data that might be generated in HSMP, the equipping of a typical hot strip mill with IIoT devices can lead up to several million sensor data each month, which all must be stored and processed. If two or more faults occur sequentially or simultaneously in HSMP, experts and third-party service providers can detect the anomaly behavior and operating conditions of the production process by analyzing the stored sensor data. Hence, as a precaution

of future production process failure and product quality, sharing and analyzing of time-series data, and subsequent detection and isolation of faults are of significant importance.

Due to storage and processing constraints of IIoT devices, nowadays, the data storage and processing functionalities are mostly shifted to the time-series database in cloud platform, e.g., Azure time-series Insight, Elasticsearch [5]. Further, processing and storing the data in the cloud platform enhances ubiquitous access, scalability and sharing possibilities. However, secure data storing in the cloud poses significant privacy risks, including unauthorized access of production line efficiency data stored in the cloud and arbitrary threats due to the cloud being compromised by a curious employee. To address privacy risks, encrypted databases have appeared as a promising solution in the recent past. The main advantage of this approach is that it allows data owners and third-party services to query encrypted data while maintaining both functionality and confidentiality. Recently, research in this domain has led to several encrypted databases, e.g., relational databases [6], graph databases [7] and batch analytics [8].

In smart manufacturing, secure time-series data storing in the cloud and sharing them with third-party services come with unique performance and security challenges that current encrypted data processing systems fail to meet. For example, query processing over time-series data at the cloud server side should support low-latency interactive queries, concurrently scale to large volumes of data, and maintain high write throughput. To address these challenges, numerous databases have been devised, particularly for time-series data [9, 10]. However, all these databases incur significant overhead during encrypted data processing. Besides, another key challenge in smart manufacturing is that privacy should co-exist during queries on data statistics, e.g., finding the standard deviation, mean, which usually indicates sharing data to be examined by third-party services. It means a comprehensive method to data security is essential for secure sharing of encrypted data. Further, data sharing must be fine-grained, as it is often unnecessary to allow third-parties free access to the data. Instead, data owners might like to (i) share only statistical computation of the data, e.g., mean, sum, max, min, (ii) restrict the granularity at which such statistical computations are reported, e.g., per-hour, per-day (iii) restrict the time interval over which queries are generated, e.g., February 2021, and (iv) a blend of the earlier three choices. Likewise, the intended level of granularity and scope of sharing can differ significantly among applications and users. Thus, we believe that support for encrypted query processing should go together with access control to restrict the scope of data that users may query. The sharing procedure for data stream stored in the time-series databases is considerably distinct from traditional databases. Particularly, in smart manufacturing, various levels of production process continuously push data streams to the cloud, where numerous services can subscribe to access and analyze data streams. Hence, the time-series databases need flexible access policies. Furthermore, often there is a requirement to aggregate and analyze time-series data from different production processes collaboratively. This indicates that we require to design an end-to-end encryption technique that is compatible with this

sharing procedure.

Storing raw data streams in ciphertext form is an effective approach for protecting data streams from data breaches and malicious cloud providers. However, a comprehensive approach for protecting data streams should also include techniques to implement access control policies, as in smart manufacturing ecosystem, data streams are usually shared with several thirdparty services. Effective access control policies will not only ensure the fine-grain data stream sharing with the third-parties, but also restrict the unnecessary exposure of data streams. Most of the existing state-of-the-art security solutions are designed for relational databases instead of time-series database [9, 10, 11, 12], where low-level access policies are adopted. Even if an access policy has been imposed for data streams in the time-series databases, the outcome of the access policy is binary, i.e., decline or grant, which is too coarse. Although, the researchers in [13, 14] have proposed a security mechanism for storing and sharing of data streams, it is vulnerable to malleability attacks [15]. Particularly, during malleability attacks, adversaries generate numerous false packets by altering ciphertext (without knowing the secret key), resulting in significant wastage of computational resources. Besides, the existing time-series databases failed to provide suitable access policies to allow data owners a fine-grain protection during selective and secure sharing of data streams with third-party services in multi-user smart manufacturing settings. An example of such policies can be a factory manager choosing to simultaneously share hourly averages of the number of faulty products produced from the factory with the production department and per-minutes averages with their quality control department, but only during morning shift.

Summary of Contributions. Our main contributions in this paper are as follows.

- We design SmartCrypt, a symmetric homomorphic encryption based access control technique for flexible and fine-grain sharing of encrypted data streams. Since our chosen symmetric homomorphic encryption scheme is additively homomorphic, it supports secure aggregation of encrypted data streams.
- We then propose an in-network false data filtering scheme to avoid malleability attacks. Particularly, we introduce a Homomorphic Message Authentication Code (HomMAC) based verification technique that supports source authentication and provide data integrity checks.
- We experimentally show the feasibility of SmartCrypt. We measure the performance of SmartCrypt in terms of overhead, latency and throughput. Our experimental results show that SmartCrypt significantly improves the overhead without compromising the latency and throughput compared to the state-of-the-art realizations, TimeCrypt and CrypSH.

Organization. We organize the rest of the paper as follows. Section II reviews the related work. In Section III, we present the system model. Section IV describes the construction of SmartCrypt in detail. In Section V, we provide a comprehensive security analysis of SmartCrypt. Section VI evaluates the performance of SmartCrypt and compares them to the state-of-the-art. Finally, Section VII concludes this work.

II. Related Work

A compendium of research on encrypted database search, privacy-preserving systems, and crypto-enforced data access can be found in the recent past works. We here briefly summarize the existing works most relevant to SmartCrypt.

Encrypted Databases. In [13], a secure cloud database system,

SHAMC, has been developed for the multi-cloud environment. SHAMC uses the philosophy of secure multi-party computation and homomorphic encryption for storing data and executing queries on the ciphertext. Burkhalter et al. [14] designed a data privacy system, Zeph, that allows users to enforce access policies while sharing and processing data with thirdparty services. Zeph employs additive homomorphic encryption method to securely share and store data streams. To support encryption and compression simultaneously, an encrypted data storage system, TinyEnc has been designed by Qi et al. [16]. TinyEnc utilizes order-revealing encryption scheme and symmetric searchable encryption scheme to support secure query processing. To enable fine-grained access to the compressed data store, TinyEnc builds an innovative key-value storage structure using the idea of horizontal-vertical division. The experimental results reveal that TinyEnc significantly reduces communication overhead in data queries. Recently, Dauterman et al. [17] presented Waldo, a time-series database with several functionality and security guarantees. Waldo allows multi-predicate querying and provides malicious security. Waldo leverages function secret sharing, a recent cryptographic tool for secure sharing. All the existing encrypted database systems have ensured authentication and confidentiality. However, none of the systems support cryptographic access control except Zeph, as in the case of SmartCrypt. Moreover, existing systems are inefficient enough to store and share continuously generated large-scale time-series data. Although Zeph enables users to cryptographic access control, it is inefficient for interactive queries on massive time-series data. Closest to our work is TimeCrypt [18], which envisions a scalable time-series data management and multi-party computation platform for sharing with third-party services. TimeCrypt uses a partially homomorphic encryption-based access control method to achieve access control. Further, TimeCrypt uses hash functions as a Pseudo-Random Generator (PRG) to ensure the integrity.

Cryptography-based Data Access. In a different line of work, researchers have investigated how to securely store the encrypted data streams, while allowing query processing and sharing operation to third-party services over the encrypted data streams [19, 20, 21]. For secure searching and sharing of industrial data, Zhang et al. [19] proposed a lattice-based public key encryption technique. The proposed technique has been further extended to avoid insider keyword guessing attack. The proposed technique is provable secure and achieves forward security. One of the main limitations of the proposed technique is that it does not support analytical operations. To support selective sharing, querying, and processing of data, a blockchain smart contract-based system has been designed in [20]. The system uses a tamper-proof log of the transaction on data to trace the malicious operations on data or regulation. Similar to [20], Hu et al. [21] designed a blockchain based data sharing system, Ghostor, which obscures user identity and enables users to detect server-side integrity violations. In a different work, an efficient and leakage-free encrypted data search system, DORY, has been designed by Dauterman et al. [22]. DORY uses distributed point function to hide search access patterns and Bloom filter to reduce time for the scan. Although, Ghostor and DORY provide necessary security while sharing and searching, they incur significant overhead in comparison to SmartCrypt. For fine-grained accessing and sharing of data, CrypSH have been designed in [23]. CrypSH uses a somewhat homomorphic encryption technique, which allows third-party services to perform algebraic operations for evaluating the multi-variate polynomials on encrypted data. However, CrypSH is

inefficient to rotate large number of keys for ensuring flexible access control over continuously generated time-series data. More recently, Zhang et al. [24] proposed a secure revocable fine-grained access control and data sharing scheme for supervisory control and data acquisition system in IIoT. The proposed data sharing scheme uses elliptic curve encryption for data encryption and fine-grained data transfer in a cloud server environment. A promising solution to achieve data privacy protection and access control in data sharing is Attribute-Based Encryption (ABE) [25, 26]. Although ABE can impose complex access control to encrypted data, it suffers from expensive crypto operations and overheads increase linearly with the number of attributes.

Secure Aggregation Statistics. Recently, secure aggregation techniques have been employed in several system designs, mainly to allow third-party services to perform statistics over users' data without gaining access to individual data [27, 28, 29]. In one such work, Roth et al. [27] designed a system for secure statistical analysis without compromising user privacy. They used multi-party computation and homomorphic encryption for their designed system. Similar to [27], a decentralized system has been introduced for privacy-conscious statistical analysis on distributed datasets in [28]. For data confidentiality and user privacy, they used homomorphic encryption and differential privacy, respectively. Niu et al. [29] proposed a framework based on dependent differential privacy for performing aggregate statistics over private correlated data. The proposed framework enhances the utility of aggregate statistics and guarantees arbitrage freeness compared to the conventional differential privacy based techniques. In a different approach, Wang et al. [30] proposed PANDA, a lightweight secure aggregation scheme for resourceconstrained devices. To bypass the trusted entities requirement and reduce overhead, PANDA uses trusted execution environment. Additionally, PANDA integrates the certificate-aided function authorization to prevent leakage from unauthorized functions. To ensure the correctness of aggregation results, Yan et al. [31] proposed a verifiable, reliable and privacy-preserving data aggregation scheme. The proposed scheme utilizes Paillier encryption for protecting the data privacy. To reduce the overhead and improve scalability, the work proposed two-tier aggregation architecture. Compared to SmartCrypt, all these approaches [27, 28, 29, 30, 31] need data producers to proactively participate in the aggregation procedure and should keep data local.

- Monitoring, analyzing and visualizations. Data consumers depend on data owners to access data streams. Specifically, in SmartCrypt, the data stream is end-to-end encrypted at the data producer and every data consumer determines the corresponding decryption key based on the encrypted authorization token (see Section 4.3.3).
- Data Owner. It owns the data stream and grants access permissions to its generated data stream. In SmartCrypt, data owners determine policies to selectively expose their data streams to data consumers. Based on the defined access policy, the database server grants or denies data stream access requests.
- Database Server. It is mainly responsible for storing encrypted data stream generated by the data producers. It is also responsible for executing statistical and analytical queries received from data consumers on encrypted data. Moreover, database server allows data consumers to execute queries and access results only based on policies defined by the data owner. In our work, access permissions are cryptographically bound to a particular owner's identity, e.g., public key.

In SmartCrypt, the cloud server performs analytical and statistical queries over the ciphertext (i.e., without requiring to decrypt the raw data) and sends back the ciphertext to the data consumer. Only the data consumer, which owns the correct keys can decrypt the ciphertext and obtain statistical result (e.g., mean/max/min), analytic (e.g., trend detection), and life cycle operation (e.g., data summarization). To augment fast queries and analytics, SmartCrypt creates in-memory encrypted indices.

III. Threat Model

In this work, we consider three most prominent threats in the cloud-based IIoT setting, namely, network-based attacks, curious Data Base Administrator (DBA) and cloud server compromise. The adversary model considered in this work is as follows.

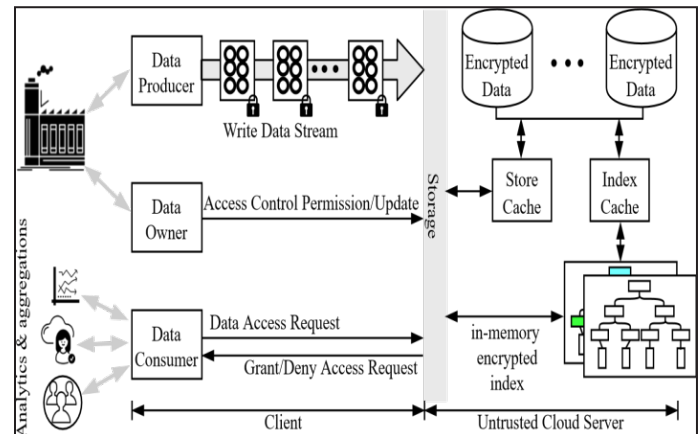


Fig. 1: The SmartCrypt architecture.

1. Network-based attack. We assume that attackers target the communication between the cloud server and IIoT devices. We also assume that attackers are capable of performing passive or active attacks to reconstruct the data owners time-series data. For example, a passive attacker can perform traffic analysis, replay and eavesdrop attacks. Likewise, an active attacker can perform impersonation or man-in-the-middle attacks.
2. Curious DBA. We consider that the DBA is honest but curious and launches passive attack. In particular, we assume that a DBA is capable of correctly executing the database protocols. However, a DBA is neither capable of altering stored time-series data nor query results, as it would malign the reputation of the cloud service provider. In our work, a curious DBA can launch passive attacks including Database Management System (DBMS) software compromises, random access memory of physical machine and root access to DBMS machines.
3. Cloud server compromise. We assume that an adversary is capable of compromising the cloud server and reveal all the encrypted time-series data stored in the cloud. As an attacker gains access to the cloud server, it can perform several attacks, including Denial-of-Service (DoS) attacks.

A. Design and Security Goals

The goals of SmartCrypt are to provide a basic security solution during storing of time-series data in the cloud and defend DoS attacks during retrieving and sharing time-series data with third-party services. Considering the typical requirements of a cloud-based IIoT system, the basic security goals of SmartCrypt are as follows.

- **Authentication.** SmartCrypt guarantees the authenticity of the IIoT devices by ensuring that data streams are coming from a trustworthy source. In SmartCrypt, we use HomMAC [32, 33] to validate the authenticity of the IIoT devices.
- **Integrity.** SmartCrypt guarantees the integrity by ensuring that correct data streams are received from the cloud server. Particularly, SmartCrypt guarantees that the data consumer should be able to detect any modification of data stream during communication with the cloud server. Further, by ensuring integrity, SmartCrypt guarantees that a malicious cloud server cannot modify the computation, other than denying service.
- **Confidentiality.** To ensure the confidentiality of shared data streams, SmartCrypt protects the data streams stored in the cloud server from both external and internal adversaries. In SmartCrypt, we encrypt data streams using semantically secure encryption before it leaves the data producer. As SmartCrypt never reveals the decryption keys to the cloud service provider, confidentiality is guaranteed even in case of a cloud server compromise.

Guarantees. As a security measure, our primary objective is to guarantee the confidentiality and integrity of computations running on the cloud server. Alternately, SmartCrypt guarantees that an adversary cannot read and tamper with data stored in the cloud server and manipulate query execution. Additionally, SmartCrypt guarantees that entities can securely share the data among themselves. To support secure sharing, SmartCrypt leverages the public-key infrastructure, where an entity is identified with a private and public key-pair. Besides, SmartCrypt does not guarantee access control collusion resistant, particularly during resolution-based sharing and interval sharing. For example, an entity with access to aggregate time-series data over the time intervals $[t_1, t_3]$ and $[t_2, t_3]$ can easily determine the aggregated time-series data over the overlapping range $[t_1, t_2]$ by determining the difference. As cryptographic techniques are insufficient to prevent collusion, hence it is the responsibility of the data owner to grant access permission carefully while sharing different resolutions over overlapping time intervals. Moreover, SmartCrypt guarantees collusion resistance and improved performance while sharing time-series data over the non-continuous time intervals.

B. Assumption

This work assumes that the cloud is honest but curious, data consumer is semi-trusted, data owner and data producer are fully-trusted. We also assume that every data producer reports correct time-series data, and performs data serialization and encryption accurately. SmartCrypt guarantees confidentiality, integrity protection and data authenticity as data are end-to-end encrypted and without the private key the raw data cannot be accurately retrieved. Furthermore, we assume that an attacker does not collude with the actors that possess the private key, since cryptographic techniques alone cannot protect this type of attack. Finally, we assume that SmartCrypt behaves correctly in normal scenario and does not disclose private key to adversaries.

IV. The Proposed Scheme: SmartCrypt

This section provides the details about our proposed scheme. Particularly, Section 4.A presents an overview of SmartCrypt. In Section 4.B, we discuss the processing and storing mechanisms of encrypted data. Finally, we put forward a sharing mechanism in Section 4.C.

A. SmartCrypt in a Nutshell

At a high level, SmartCrypt is a new time-series data protection scheme, which stores the encrypted data streams on the cloud, while allowing for secure query processing over the encrypted data streams and fine-grain sharing of data streams with principals. We introduce a symmetric homomorphic encryption-based access control technique that supports both computations over encrypted data streams and fine-grained access control. Precisely, SmartCrypt allows data owners to cryptographically restrict a data consumer Alice to query encrypted data at a well-defined temporal range and granularity, while at the same time allowing a data consumer Bob to perform queries on the same data at a distinct granularity without introducing (i) data redundancy, and (ii) significant delays. In the existing works, usually secure data storing and sharing have been addressed independently, where the former is addressed through encrypted data processing, and the latter through cryptographically enforced access control. However, SmartCrypt facilitates both while meeting the access control and performance requirements of time-series data streams.

In SmartCrypt, the key idea behind the symmetric homomorphic encryption-based access control is based on the study that time-series data are continuous in nature, and time is the key attribute for processing and accessing of data streams. Therefore, we divide time-series data streams into chunks, where each chunk corresponds to a fixed-length time segment, and encrypt each chunk with a unique key employing symmetric homomorphic encryption. To process queries, the cloud server calculates the aggregate function on the encrypted chunks. By doing this, SmartCrypt facilitates aggregation-based statistical queries, e.g., mean, sum or transforming queries to be aggregation based, e.g., regression, max/min. Further, this enables SmartCrypt to define fine-grained access policies at the stream segment granularity. This essentially introduces two challenges: (i) how to generate and manage a large number of unique keys in an efficient and scalable fashion, and (ii) how to express data stream access policies succinctly that is then used to compute decryption keys associated with the access policy. To address these challenges, similar to [18], SmartCrypt associates keys with temporal segments. This helps SmartCrypt to avoid the requirement of maintaining a mapping between keys and ciphertexts. To compute these keys, SmartCrypt builds a hierarchical key derivation tree that enables us to devise fine-grained access policies over stream data and share keys efficiently.

B. Storing of Encrypted Data

In this section, we illustrate how SmartCrypt encrypts and stores the time-series data streams in the cloud server.

1. Symmetric Homomorphic Encryption

In this section, we present our symmetric homomorphic encryption-based access control technique in detail. Let m_i be a message to be encrypted from the message space $[0, M - 1]$, i.e., $m_i \in [0, M - 1]$ and size of m_i is an integer, where $i = 1, \dots, n$. Let k_i be a randomly generated secret key stream used to encrypt m_i , where $k_i \in [0, M - 1]$. To encrypt m_i , the data owner computes ciphertext as $c_i = E_{k_i}(m_i) = (m_i + k_i) \bmod M$. In contrast, to retrieve m_i for given k_i , one can perform decryption as $m_i = D_{k_i}(c_i) = (c_i - k_i) \bmod M$. This scheme is semantically secure once the key streams are pseudorandom and none of the key is reused [15]. Since our encryption is based on a symmetric homomorphic encryption, we can easily decrypt the aggregated ciphertext for the given aggregated secret key streams as follows:

$$\begin{aligned}
 & \prod_{i=1}^n X^{m_i} = \prod_{i=1}^n (DP^{n_i-1} k_i^{-1} X^{c_i}) \\
 & X^{\sum_{i=1}^n m_i} = \prod_{i=1}^n (c_i^{-1} k_i) \pmod{M}. \tag{1}
 \end{aligned}$$

It is worth noting that, the correctness of Eq. (1) is assured once M is sufficiently large. Thus, we choose M as 264 [18], where 0 ≤ m < M. We notice from Eq. (1) that the computation overhead to aggregate secret keys is linear and depends on the number of aggregated ciphertexts. It signifies that a data consumer has to perform the same amount of computation as the cloud server for aggregating secret keys. Since the timeseries data streams are usually aggregated over a continuous time range, we employ an optimized aggregation strategy similar to the one used in [8] to reduce this linear overhead to a constant. Specifically, in our optimized aggregation strategy, we select the individual encryption keys in such a way that the inner keys cancel out each other during aggregation. Suppose, for instance, Alice wants to share with Bob an encrypted data stream. Alice can do this by replacing the individual key for each data stream with a composite key that links subsequent data streams. In SmartCrypt, we do this as follows:

$$E_{k_i'}(m_i) = (m_i + k_i') \pmod{M}, \tag{2}$$

where k_i' is a composite key and k_i' = k_i^{-1} k_{i+1}. Following the same strategy, to decrypt in-range aggregated ciphertexts as given in Eq. (1), we just need the two boundary keys as shown below:

$$\begin{aligned}
 X^{k_i'} &= \prod_{i=1}^n (k_{i+1}^{-1} k_i) + (k_2^{-1} k_3) + \dots + (k_{n-1}^{-1} k_n) \\
 &= k_1^{-1} k_n. \tag{3}
 \end{aligned}$$

From Eq. (3) it is clear that due to our optimized aggregation strategy, the decryption time is independent of in-range aggregated ciphertext in SmartCrypt. It is worth noting that, SmartCrypt enjoys standard semantic security even with the introduction of optimized aggregation strategy, as an adversary cannot employ the key canceling property without access to the key streams.

2. Malleability

As mentioned in Section 1, homomorphic encryption schemes are inherently susceptible to malleability attacks. Particularly, in malleability attacks, an adversary alters a ciphertext by injecting false data. For instance, an adversary can manipulate (m + k) mod M to (m + k) + θ mod M, where θ is a false data. Considering the application scenario of SmartCrypt, such an attack can be initiated from a dishonest cloud server by duplicating or manipulating ciphertext, resulting in erroneous query results. To mitigate malleability attacks, we design a verification technique that enables the cloud server to authorize the outcome of in-range aggregations over ciphertexts with a succinct authentication tag. In this work, we use HomMAC to generate the authentication tag. In our HomMAC, the data owner generates a tag σ_i for every ciphertext c_i, where i = 1, ..., n. We employ a strategy for generating σ_i similar to the one proposed in [34]. Particularly, we define σ_i as follows:

$$\sigma_i = (r_i - c_i)q \pmod{p}, \tag{4}$$

where r_i is a key for ciphertext c_i and it is computed using a pseudorandom function f: {0,1}*, q is the HomMAC key, and p is a prime number. In SmartCrypt, the data owner computes aggregations on both tags using Eq. (4) and ciphertexts, i.e., $\prod_{i=1}^n (c_i, \sigma_i) = (c_{agg}, \sigma_{agg})$, and uploads in the cloud server. A data consumer in possession of HomMAC decryption key verifies the received data stream by checking the received (c_agg, σ_agg) as follows:

$$\begin{aligned}
 & \prod_{i=1}^n X^{c_i} X^{\sigma_i/q} \\
 & r_i = c_i + \sigma_i/q \pmod{p} \quad i=1 \dots n \\
 & = c_{agg} + \sigma_{agg}/q \pmod{p}. \tag{5}
 \end{aligned}$$

If Eq. (5) holds, it means that n tags are valid. We next determine the computational complexity of our HomMAC technique, it is O(n).

3. Integrity of HomMAC

Although our HomMAC gives the necessary security, it incurs a verification overhead, which is linear with the number of records in the aggregation query. Hence, we use similar key canceling approach as discussed above with respect to encryption. More specifically, we define a HomMAC key stream {h_1, h_2, ..., h_n} for each ciphertext c_i, the data owner calculates the HomMAC tag σ_i as follows:

$$\begin{aligned}
 \sigma_i &= (h_i' - c_i)q \pmod{p} \\
 &= (h_i - h_{i-1} + 1 - c_i)q \pmod{p}, \tag{6}
 \end{aligned}$$

where h_i' = h_i - h_{i+1} and i = 1, ..., n. By setting h_i' = h_i - h_{i-1} + 1, it allows us to verify the HomMAC tag at a constant time irrespective of input size. Further, Eq. (6) indicates that we need only two outer keys during verification:

$$\begin{aligned}
 & \prod_{i=1}^n X^{h_i'} = h_1 - h_n + 1 \\
 & = c_{agg} + \sigma_{agg}/q \pmod{p}. \tag{7}
 \end{aligned}$$

It is worth noting that, inner key cancellation idea in Eq. (7) is the main support for our cryptographic access control. Further, our HomMAC technique integrates suitably with our goal of fine-grain access control.

C. Sharing of Encrypted Data

In SmartCrypt, our objective is to allow data streams access permissions to the third-party services at arbitrary intervals like from 13.00-Feb 05 till 12.00-Feb 06 2021 for restricting the unnecessary exposure of data stream. To achieve this, SmartCrypt partitions the data streams into fixed-length time segments or chunks of size Δ, e.g., 20 sec. Each chunk is then encrypted using a separate key from the key stream, subsequently indexed by the time window of the chunk. The standard practice for efficiently generating such large key streams is to exploit a pseudorandom function with a pre-shared secret key. This enables anyone to generate a large key stream for a particular secret key. However, this approach is not efficient for sharing data streams of arbitrary intervals or to restrict access permission to temporal data, e.g., daily or hourly summaries. To achieve finegrain access control and enable data owners to share encrypted data streams of desired intervals, we devise a hierarchical key derivation tree.

1. Key Derivation Trees

Our construction of key derivation tree is based on the GGM tree [35]. Generally, GGM tree defines a balanced binary tree over the PRF, where each node contains a unique pseudorandom value. We derive the key stream $\{k_1, k_2, \dots, k_{2^l-1}\}$ from the leaf of GGM tree as shown in Figure 2. We contrast the key derivation tree from a secret random seed as the root. We then generate the child or internal nodes using a PRG, which takes the parent string as the input. We define a PRG for building GGM tree as follows: $G : \{0,1\}^\lambda \rightarrow \{0,1\}^{2\lambda}$. Our PRG comprises $G_0(x)$ for the left-hand child and $G_1(x)$ for the right-hand child, where

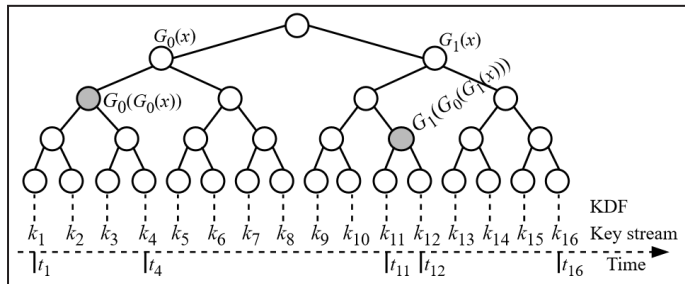


Fig. 2: Key derivation tree in SmartCrypt.

x is the parent node whose length of PRF value is λ . We then define a PRF as: $F = \{ f_x : \{0,1\}^s \rightarrow \{0,1\}^\lambda \}_{x \in \{0,1\}^\lambda}$, such that $f_x(b_{s-1} \dots b_0) = G_{b_0}(\dots(G_{b_{s-1}}(x)))$, where s is polynomial in λ and $b_{s-1} \dots b_0$ is the input bit stream of size s . We recursively utilized the aforementioned PRF to build the key derivation tree of desired depth d . Since in the smart manufacturing domain data streams are often generated at high-frequency, we choose a large l to generate virtually infinite key streams to encrypt chunks, e.g., one chunk per second.

SmartCrypt encrypts each chunk using a unique key computed from the key derivation tree. We construct the key derivation tree in such a way that the data owner can selectively share data streams at any arbitrary time interval; to derive the corresponding keys, only sharing with the inner nodes in the key derivation tree is required. For example, in Figure 2, given the highlighted inner node with PRG $G_0(G_0(x))$ a data consumer can access data stream of interval $[t_1, t_4]$ and can derive the corresponding decryption keys $\{k_1, k_2, k_3, k_4\}$. Likewise, a data consumer can access data stream of interval $[t_{11}, t_{12}]$ and can derive the corresponding decryption keys $\{k_{11}, k_{12}\}$ as shown in Fig. 2. Although our key derivation tree enables a data owner to selectively share a data stream of any arbitrary time interval, this procedure lacks sharing continuous data streams with data consumers. To overcome this limitation, we introduce a key regression technique for sharing decryption keys. Specifically, a data owner generates the necessary decryption keys through a token considering the access permission of the data consumer.

2. Key Regression

A key regression technique allows efficient sharing of keys for continuous data streams. Our key regression technique is built with the PRG used in the key derivation tree. Particularly, in SmartCrypt, a data owner shares the PRG of some inner tree nodes combined into a token instead of sharing the individual key for each chunk. Suppose, for instance, a data owner allows a data consumer to access data stream of interval $[t_1, t_4]$ and corresponding decryption keys $\{k_1, k_2, k_3, k_4\}$. In SmartCrypt, instead of sharing the chunk key-by-key, the data owner shares the PRG of a single node, i.e., $G_0(G_0(x))$ with the data consumer. Given the PRG

of a single node, a data consumer can derive decryption keys $\{k_1, k_2, k_3, k_4\}$ by applying the PRG successively. It is worth noting that, determining the PRGs of parent, sibling, or any ancestor nodes is computationally not feasible due to one-way property of PRGs. Therefore, data consumers cannot derive any key, except the chunks, they are allowed to access.

3. Token Distribution

After defining the data access policy for sharing by the data owner, SmartCrypt creates a token, which encapsulates the PRG of the inner nodes of the key derivation tree required for determining the respective shared key streams. SmartCrypt uses the same key derivation tree during encryption (or encapsulation) and HomMAC key streams, however, with a separate Key Derivation Function (KDF). In SmartCrypt, the token further includes encoded information about the tree depth. Finally, the token is encrypted with the data consumer's public key and stored at the cloud server. The data consumer retrieves the encrypted token from the cloud server to decrypt the query results or data. We considered ABE [36] to encrypt the token. Although, we did not consider any key update strategy in SmartCrypt, as a possible solution, the data owner can update the token to revoke or update data stream access.

4. Resolution-based Sharing

As mentioned earlier, SmartCrypt enables a data owner not only to limit the data stream access to a time range for a data consumer, but also defines the temporal granularity like per hour at which they can query or retrieve data. SmartCrypt defines the maximum resolution for a query and access permission through the chunk size Δ . To limit access to data at several resolution levels, SmartCrypt exploits the keys cancel out procedure during in-range aggregation, as presented in Section 4.2.1. Typically, a ciphertext derived in SmartCrypt via an in-range aggregation over the time interval $[t_i, t_j]$ as: $P_y^{i=i} c_y = P_y^{j=i} (m_y + k_i - k_j) \text{ mod } M$, where the inner keys from k_{i+1} to k_{j-1} are cancelled out. Therefore, a data consumer can decrypt the aggregated data with the help of two boundary keys k_i and k_j in spite of without accessing of the individual ciphertext. As we assume SmartCrypt shares data of chunk size Δ , the selection of resolution levels should be multiples of chunk size Δ . For example, if a data owner desires to limit access to a 4-fold resolution of the chunk size, the data consumer will receive only $\{k_1, k_5, k_9, \dots\}$ from the data owner. Based on the received boundary keys, the data consumer can decrypt the aggregated ciphertexts at 4Δ or lower resolutions. However, the data consumer cannot decrypt the aggregated ciphertexts of higher resolutions due to the unavailability of inner keys.

D. Data Analytics and Processing

To achieve the goals of storing and sharing time-series data, SmartCrypt must efficiently deal with a significant amount of data without incurring latency. In SmartCrypt, we address this challenge by devising server side encrypted indices and client side serialisation approaches.

Server Side Encrypted Index. In SmartCrypt, the server side retains an in-memory encrypted index in the form of an aggregation tree over the encrypted data stream. This is an essential building block that allows SmartCrypt to execute analytics at high speed on massive encrypted data streams and allows effective data preservation. The in-memory encrypted index structure in SmartCrypt is a k -ary tree, where every node keeps k statistical summaries of the sub-tree under it. The tree leaves typically store the chunk digests encrypted with our proposed symmetric homomorphic encryption-based

access control technique at the client side. Upon receiving a new chunk digest, the server includes it as a leaf node in a k-ary tree and revises the parent node's statistical summaries by executing an encrypted aggregation. As SmartCrypt maintains a statistical index for time-series data, the server circumvents the requirement of serial scanning, resulting in significant improvement in response time for statistical queries.

Client Side Data Serialization. SmartCrypt partitions the data streams into chunks of predefined fixed-length time intervals and it is a popular approach for time-series data. Data chunks comprising raw data and digests are serialized and encrypted at the client side. In SmartCrypt, we set the content of a digest per stream depending on the supported queries. SmartCrypt supports sum, mean and count as default queries. Additionally, SmartCrypt can support other query types like standard deviation, variance, max/min.

Data Analytics. Similar to TimeCrypt, SmartCrypt allows various forms of machine learning enabled analytics, such as aggregation-based encoding. Such types of analytics are frequently used in time-series data to understand and find various runtime anomalies, patterns and trends. Further, SmartCrypt allows aggregatable transformation operations, including Sketch [37]. An example of such analytics can be used for identifying a common factory operational tendency over a defined time interval. It enables factory operational manager to assess the magnitude of an operational trend and co-related tasks for detecting possible runtime anomalies. A simple, still powerful technique for operational trend detection is linear regression using least square. To determine a linear regression over a data stream, we can define the per chunk digest as $(P_{n_i=1} x_i, P_{n_i=1} t_i x_i)$. In this way, SmartCrypt can perform expensive aggregation operations at the server side. Since the ciphertext is used in the training phase, such learning on aggregated data provides further privacy gains.

Performance Evaluation
This section reports results from our experimental evaluation of SmartCrypt.

V. Evaluation Setup and Dataset

We built a prototype implementation of SmartCrypt in Java. During implementation, similar to TimeCrypt, we use protobufs protocol and Netty for client-server communication and network communication, respectively. To expedite index node access, SmartCrypt augments the encrypted index with the inmemory cache caffeine [38]. For the storage back end, our prototype implementation uses Apache Cassandra. We conduct experiments in InfluxDB, on M5 instances equipped with a 4.9 GHz CPU running Ubuntu 18.04. SmartCrypt's clients run on several m5.2xlarge instances. Whereas, SmartCrypt's server runs on a m5.xlarge instances. We connect both the client and server in the same data center network, with an average transmission delay of upto 10 μ s. For our micro-benchmark, we measure the overhead of encryption and decryption on end IIoT devices. For the source of sensitive time-series data, we assume resource-constrained IIoT devices with similar specifications as OpenMotes [39]. We use the publicly available ECO dataset [40] for performance demonstration. ECO dataset contains anonymized power demand data collected over 8 months and sampled at 1 HZ rate. During data collection, every smart meter uploads a data chunk every 5 s, however, the service can calculate per day aggregate for each data stream. We measure the overhead of SmartCrypt and compare it with (i) our main competitor TimeCrypt [18], where authors used a homomorphic

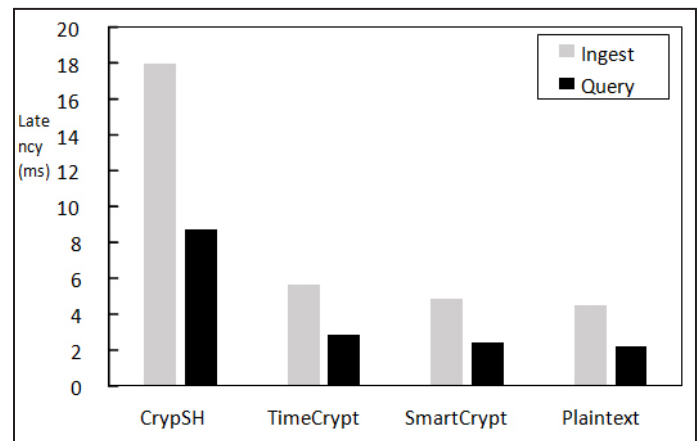


Fig. 5: Latency measurement for ingest and statistical queries with $\Delta = 10$ s and 50Hz data rate.

involves higher cost associated with the key derivation tree than SmartCrypt. Whereas, CrypSH requires to solve the discrete logarithm using Pollard's lambda algorithm, which is computational intensive. We notice from Table 3 that SmartCrypt is 1.8 \times faster than TimeCrypt while performing encryption/decryption operations in IIoT device. In contrast, SmartCrypt is 1.5 \times faster than TimeCrypt while performing HomMAC operation. Now, if we compare the performance between SmartCrypt and CrypSH, the encryption/decryption time of CrypSH is significantly higher than SmartCrypt. Likewise, we notice that SmartCrypt provides superior performance compared to TimeCrypt and CrypSH while measuring crypto operations in Laptop.

Table 3: Comparison of crypto operations on IIoT devices (OpenMote) vs. laptop

	[Enc/Dec]	[HomMAC]	[Enc/Dec]	[HomMAC]
IIoT	5.17ms	102.93 μ s	2.84ms	68.35 μ s
Laptop	27.6 μ s	1.51 μ s	15.72 μ s	0.94 μ s

1. SmartCrypt Performance: Scalability

This section measures the scalability of our symmetric homomorphic encryption-based access control technique in terms of storage, communication and computation overheads. We use overheads for scalability measurement as the scalability of SmartCrypt significantly relies on determining HomMAC, encryption key and cost of key distribution. While measuring the overheads, we assume that a data owner contains 500 data streams and a subset of each data stream is shared with a data consumer. During scalability measurement, we consider a data stream consisting of 5 s chunk intervals over 1 year with resolution as hour/day/month.

Storage Overhead. In SmartCrypt, storage overhead comprises (i) key storage at the data consumer (or, client side), and (ii) key storage at the server side. Table 4 shows the average storage overhead of SmartCrypt, TimeCrypt and CrypSH. We notice that SmartCrypt requires 0.53 MB memory for key storage at the client side. Whereas, memory needs for key storage in TimeCrypt and CrypSH is 0.66 MB and 2.01 MB, respectively. Therefore, SmartCrypt reduces the storage overhead at the client side by 25% and 279% than TimeCrypt and CrypSH, respectively. The key storage at the client side is predominantly associated with the resolution-based sharing operation for both SmartCrypt and TimeCrypt. Whereas for CrypSH, the client side key storage is linked with the third-party sharing operation. It is worth noting

that the key storage for the resolution-based sharing operation increases linearly with time in both SmartCrypt and TimeCrypt. We observe from Table 4 that SmartCrypt needs 0.64 MB memory for storing keying material per stream at the server end, whereas, TimeCrypt and CrypSH require 0.85 MB and 2.86 MB memory, respectively. Thus, SmartCrypt reduces the storage overhead by 24.70% than TimeCrypt and 346% than CrypSH at the server side.

Table 4: Average storage overhead

Scheme	Operation	Size [MB]
CrypSH	Key storage at client	2.01
	Key storage at server	2.86
TimeCrypt	Key storage at client	0.66
	Key storage at server	0.85
SmartCrypt	Key storage at client	0.53
	Key storage at server	0.64

Communication Overhead. Table 5 shows the average communication overhead of SmartCrypt, TimeCrypt and CrypSH. We observe from the table that SmartCrypt requires 0.58 MB memory for sharing 1 year of data as considered in our experimental scenario. On the contrary, TimeCrypt and CrypSH require 0.76 MB and 2.49 MB memory, respectively for the same purpose. Specifically, SmartCrypt reduces the communication overhead by 23.68% and 329% than TimeCrypt and CrypSH, respectively.

Table 5: Average communication overhead

Scheme	Overhead [MB]	CI [MB]
CrypSH	2.49	[2.32, 2.63]
TimeCrypt	0.76	[0.70, 0.81]
SmartCrypt	0.58	[0.53, 0.64]

Computation Overhead. We here discuss the computation overhead of SmartCrypt, TimeCrypt and CrypSH, as summarized in Table 6. One of the major computations in SmartCrypt is determining the access token. We notice from the table that SmartCrypt outperforms both TimeCrypt and CrypSH in terms of computation overhead. Specifically, we observe that

Table 6: Average computation overhead

Scheme	Operation	Overhead	CI
CrypSH	Access token	32s	[21s, 43s]
	Decryption key	18k/s	[9k/s, 27k/s]
TimeCrypt	Access token	78ms	[71ms, 83ms]
	Decryption key	209k/s	[198k/s, 220k/s]
SmartCrypt	Access token	53ms	[48ms, 57ms]
	Decryption key	262k/s	[250k/s, 270k/s]

SmartCrypt, TimeCrypt and CrypSH need 53 ms, 78 ms and 32 s, respectively, for computing the access tokens of all data streams. Therefore, SmartCrypt requires 32.05% less computation than TimeCrypt. In contrast, SmartCrypt needs significantly less computation than CrypSH. In addition to access token computation, SmartCrypt needs to derive the decryption keys. We observe that SmartCrypt derives decryption keys at a rate of 262 k/s. Whereas, TimeCrypt and CrypSH compute decryption keys at a rate of 209

k/s and 18k/s, respectively. Summarily, SmartCrypt improves the decryption key computation rate by 20.23% over TimeCrypt. Compared to CrypSH, SmartCrypt improves the decryption key computation rate by 13×.

VII. Conclusion

Secure and fine-grain sharing of the time-series data with the third-party services for performing analytics is of utmost significance in IIoT to improve monitoring and maintenance. Keeping these in mind, this paper proposed SmartCrypt, a data storing and sharing system that supports scalable analytics over massive encrypted time-series data. We introduce a novel symmetric homomorphic encryption-based access control technique tailored for time-series data. SmartCrypt enables the execution of analytics over encrypted data streams and empowers users to impose access restrictions on encrypted data streams, considering their access control and privacy preferences. Our evaluation on real-world datasets show the feasibility of SmartCrypt as an authorization service for secure and fine-grain sharing, and performing analytics on large-scale time-series data.

In the future, we envision to explore innovative encryption technologies, e.g., predicate encryption to support richer range queries over encrypted data, e.g., fuzzy query and verifiable queries. Further, we plan to investigate how to revoke or update keys for high-volume and high-velocity data streams.

References

- [1] M. Centenaro, C. E. Costa, F. Granelli, C. Sacchi, L. Vangelista, A survey on technologies, standards and open challenges in satellite iot, *IEEE Communications Surveys & Tutorials* 23 (2021) 1693–1720.
- [2] Y. Wu, Z. Wang, Y. Ma, V. C. Leung, Deep reinforcement learning for blockchain in industrial iot: A survey, *Computer Networks* 191 (2021) 108004.
- [3] A. Corallo, M. Lazoi, M. Lezzi, Cybersecurity in the context of industry 4.0: A structured classification of critical assets and business impacts, *Computers in Industry* 114 (2020) 103165.
- [4] L. Ma, J. Dong, K. Peng, A novel hierarchical detection and isolation framework for quality-related multiple faults in large-scale processes, *IEEE Trans. on Industrial Electronics* 67 (2020) 1316–1327.
- [5] A. Bader, O. Kopp, M. Falkenthal, Survey and comparison of open source time series databases, in: *Proc. of Business, Technologie and Web (BTW)*, 2017, pp. 249–268.
- [6] S. Tu, M. F. Kaashoek, S. Madden, N. Zeldovich, Processing analytical queries over encrypted data, *Proc. of VLDB Endowment* 6 (2013) 289–300.
- [7] X. Meng, S. Kamara, K. Nissim, G. Kollios, GreCs: Graph encryption for approximate shortest distance queries, in: *Proc. of 22nd ACM CCS*, 2015, pp. 504–517.
- [8] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, S. Badrinarayanan, Big data analytics over encrypted datasets with seabed, in: *Proc. of 12th USENIX OSDI*, 2016, pp. 587–602. [9] Microsoft Azure, Azure Time Series Insights, [Online]: Accessed on February 06, 2021. <https://azure.microsoft.com/en-us/services/time-seriesinsights/>.
- [9] InfluxDB, InfluxDB Cloud, [Online]: Accessed on February 06, 2021. <https://www.influxdata.com/>.
- [10] Amazon, Amazon Timestream, [Online]: Accessed on February 06, 2021. <https://aws.amazon.com/timestream/>.

- [11] F. Lautenschlager, M. Philippsen, A. Kumlehn, J. Adersberger, Chronix: Long term storage and retrieval technology for anomaly detection in operational data, in: Proc. of 15th USENIX FAST, 2017, pp. 229–242.