TimeCrypt: Encrypted Data Stream Processing at Scale with Cryptographic Access Control

Lukas Burkhalter, *ETH Zurich;* Anwar Hithnawi, *UC Berkeley, ETH Zurich;*
Alexander Viand and Hossein Shafagh, *ETH Zurich;* Sylvia Ratnasamy, *UC Berkeley*

TimeCrypt: Encrypted Data Stream Processing at Scale with Cryptographic Access Control

Lukas Burkhalter[1], Anwar Hithnawi[1,2], Alexander Viand[1], Hossein Shafagh[1], Sylvia Ratnasamy[2]

[1]*ETH Zürich*   [2]*UC Berkeley*

Abstract

A growing number of devices and services collect detailed *time series* data that is stored in the cloud. Protecting the confidentiality of this vast and continuously generated data is an acute need for many applications in this space. At the same time, we must preserve the *utility* of this data by enabling authorized services to *securely* and *selectively* access and run analytics. This paper presents TimeCrypt, a system that provides *scalable* and *real-time* analytics over large volumes of encrypted time series data. TimeCrypt allows users to define expressive data access and privacy policies and enforces it cryptographically via encryption. In TimeCrypt, data is encrypted end-to-end, and authorized parties can only decrypt and verify *queries* within their authorized access scope. Our evaluation of TimeCrypt shows that its memory overhead and performance are competitive and close to operating on data in the clear.

1 Introduction

Recent years have seen explosive growth in systems and devices that collect time series data and relay it to cloud-based services for analysis. This growth is only expected to accelerate with the proliferation of IoT devices, telemetry services, and improvements in data analytics. However, with this growth has come mounting concerns over data protection and data privacy [66]. Today, the public concern over data privacy and confidentiality is reaching new heights in light of the growing scale and scope of data breaches [17, 29, 56]. To grasp the extent of this issue, one can look at the number of data breaches reported under the new GDPR *obligation to notify*, which has already exceeded 65,000 in the first year [73].

Over the last decade, encrypted databases [60, 61, 63, 75, 81] have emerged as a promising solution to tackle the problem of data breaches. The approach of keeping data encrypted while in-use allows users to query encrypted data while preserving both confidentiality and functionality. Research in this domain has led to various encrypted database designs, including designs for key-value stores [25], batch analytics [60], graph databases [53], and relational databases [63, 81]. This motivates the following natural question: *can we enable encrypted data processing for time series workloads?*

Time series workloads come with unique performance and security requirements that existing encrypted data processing systems fail to meet:

(i) Scalability and Interactivity. Query processing over time series data must simultaneously scale to large volumes of data, support low-latency interactive queries, *and* sustain high write throughput. To meet these challenges several dedicated databases have been designed for time series workloads [12, 33, 42, 45, 50, 62, 78]. A key aspect of these systems is their use of in-memory indices that store aggregate statistics, enabling faster query response times and data summarization. As we discuss in §6/§7, prior work on encrypted data processing does not easily lend itself to maintaining these in-memory indices. The overhead of the crypto primitives in encrypted data processing needs to be negligible to meet the scaling, latency, and performance requirements associated with time series workloads.

(ii) Secure Sharing. A key challenge in modern systems is that privacy must co-exist with the desire to extract value from the data, which typically implies *sharing* data to be analyzed by third-party services [54]. Hence, a truly comprehensive approach to data protection must also comprise mechanisms for secure sharing of encrypted data. Sharing should also be *fine-grained* since it is undesirable and often unnecessary to give parties unfettered access to the data. Instead, users may want to *(1)* share only aggregated statistics about the data (e.g., avg/min/max), *(2)* limit the resolution at which such statistics are reported (e.g., hourly vs. per-minute), *(3)* limit the time interval over which queries are issued (e.g., only June 2019), *(4)* or a combination of the above. Moreover, the desired granularity and scope of sharing can vary greatly across users and applications. Hence, support for encrypted query processing must go hand-in-hand with access control that limits the scope of data that users might query. The sharing paradigm in data-stream systems is distinctly different than in conventional databases. Data-stream settings feature a multitude of data sources continuously pushing data to the cloud, where various services that are often not known in advance can subscribe to consume and analyze data streams. Therefore, such systems require flexible access policies. Frequently, there is a need to fuse and analyze data from different sources collectively; this implies that we need to devise an end-to-end encryption

scheme that is compatible with this sharing paradigm.

TimeCrypt. In this paper, we present TimeCrypt, a system that augments time series databases with efficient encrypted data processing. TimeCrypt provides cryptographic means to restrict the query scope based on data owners' defined policies. With TimeCrypt, data owners can cryptographically restrict user A to query encrypted data at a defined temporal range and granularity, while simultaneously allowing user B to execute queries on the same data at a different granularity without *(i)* introducing ciphertext expansion or data redundancy, *(ii)* introducing any noticeable delays, or *(iii)* requiring a trusted entity to facilitate this.

In this work, we introduce a partially homomorphic-encryption-based access control construction (HEAC) that supports both fine-grained *access control* and *computations* over encrypted data within a *unified scheme.* These two aspects have traditionally been addressed independently: the former through cryptographically enforced access control schemes [35, 39, 49, 59, 83] and the latter through encrypted data processing [60, 63, 75, 81]. HEAC simultaneously supports both while meeting the performance and access control requirements of time series workloads. A key insight behind the design of HEAC is based on the observation that time series data streams are continuous and time is the natural attribute for accessing and processing this data. Hence, we discretize data streams into fixed-length time segments, and encrypt each segment with a different key using symmetric-key homomorphic encryption. This allows us to express fine-grained access policies at the stream segment granularity. This, however, raises two challenges; we need to manage a large number of keys in an efficient and scalable manner and translate stream access policies to the corresponding keys succinctly. To overcome these challenges in HEAC, we associate keys with temporal segments. With this, we avoid the need to maintain a mapping between keys and ciphertexts. We derive these keys from a hierarchical key-derivation tree construction that allows us to express fine-grained access policies over stream data and share keys efficiently (i.e., with logarithmic complexity).

We provide an implementation and evaluation of a prototype of TimeCrypt on top of Cassandra. We evaluate TimeCrypt in a range of scenarios combining IoT devices, AWS (for data storage and processing), and time series traces from real-world applications. We show that TimeCrypt can support a wide range of applications by developing four applications which vary in complexity and scalability requirements. Finally, we show that TimeCrypt's performance is competitive with the baseline (plaintext) and it outperforms prior work by a factor of 2 to 52 (§6). Considering an ingest workload with 5.77 million data points per second *on a single machine*, TimeCrypt's throughput is reduced only by 2.9% for both data ingest and statistical queries over encrypted data.

Contributions. In summary, our contributions are:

- We introduce HEAC, an encryption-based access control construction for stream data that is additively homomorphic. HEAC additionally provides verifiable computations over ciphertexts to ensure the integrity of the outsourced encrypted computation.

- We design, implement, and evaluate TimeCrypt, the first scalable privacy-preserving time series database that meets the scalability and low-latency requirements associated with time series workloads. We introduce a design that protects the data confidentiality, yet maintains its utility by efficiently supporting a rich set of functionalities and analytics that are key to time series data. TimeCrypt supports data lifecycle operations such as ongoing data summarization and deletion that are common in time series databases. TimeCrypt supports expressive data access and privacy policies, enforceable by encryption.

- We make TimeCrypt's code publicly available[1], both as a standalone system and as a library to be integrated with other time series databases.

2 Overview

TimeCrypt achieves its competitive performance through a careful design of cryptographic primitives tailored for time series data workloads. To understand the rationale behind our techniques, we start this section by presenting relevant background on time series data, then we give an overview of TimeCrypt, and describe our security model.

2.1 Background on Time Series Data

Time series Applications. Time series data is increasingly prevalent across a wide range of systems (e.g., monitoring, telemetry, IoT) in diverse domains such as health, agriculture [82], transportation [69], operational insight [2], and smart cities. The growth of time series data is largely attributed to the rising demand for instrumentation. Individuals and organizations are continuously logging various metrics which report the state of systems or organisms for better diagnoses, forecasting, decision making, and resource allocation. The ability to capture and analyze this data in a timely manner is key for automation and is enabling a whole new spectrum of applications [2, 33, 40, 69, 82]. The proliferation of time series data has been coupled with increasing demand for high-performance analytics over large volumes of time series, and has led to numerous designs for databases that are optimized for time series workloads [12, 33, 42, 45, 50, 62, 78].

Time series Workloads. *(i) Write and Read:* Data is append-only and typically generated at an extremely high rate (high velocity) and is initially stored at a high resolution (large volume) [12, 62]. It is not unusual for applications in this space to report hundreds of millions of data points per day [7, 12].

---

[1]Available at: https://timecrypt.io/

Hence, sustaining high read and write throughputs and scalability are key requirements when storing and processing time series data. Time is the primary dimension for accessing and processing data. Queries primarily consider temporal ranges (e.g., values from the last 3h) rather than targeting individual points. *(ii) Analytics:* Queries are primarily of aggregate and statistical nature, and specialized indices for accelerating statistical queries are common in time series databases [12, 42, 65]. Additionally, analytics of diagnostic (e.g., anomaly or trend detection) and predictive nature (e.g., forecasting) are common in this space. *(iii) Data decay:* Time series data is often machine generated, continuous, and massive. Simultaneously, the value and relevance of data decays rapidly with time. Analytics largely favor recent data over older, and roll up aggregation is commonly applied to older data to reduce storage requirements. Hence, data retention and summarization [7, 62] are crucial for these systems.

The goal of our work is to retain the performance, functionality, and scalability of existing time series databases while augmenting them with strong security and privacy guarantees.

2.2 Architecture

TimeCrypt's architecture is analogous to that of conventional times-series databases [10, 11, 12, 42, 50], where a standard distributed key-value store is extended with additional logic for time series workloads. TimeCrypt includes a trusted *client library* to realize end-to-end encryption paired with access control and integrity verification. TimeCrypt consists of two components (i.e., the client and server libraries) and involves four parties (i.e., data owner, data producer, data consumer, and database server), as illustrated in Fig. 1. A *data producer* is an entity (e.g., IoT device) that generates and uploads time series data, and runs TimeCrypt's client library which handles stream preprocessing and encryption. The *data owner* can express access permissions to its generated data. Meanwhile, *data consumers* are entities (e.g., services) that are authorized to access a user's data to provide added value, such as visualizations, monitoring, and diagnoses. TimeCrypt's server executes statistical and analytical queries directly on encrypted data. TimeCrypt supports a rich set of foundational queries that are widely used in time series workloads (§4), i.e., statistical queries (e.g., min/max/mean), analytics (e.g., prediction, trend detection), and lifecycle operations (e.g., ongoing data summarization, deletion). The server builds *in-memory encrypted indices* to support fast queries and analytics (§4).

2.3 Goals for Stream Data Access Control

Encryption is an effective tool for protecting data from external threats, breaches, or malicious providers. However, a truly comprehensive approach to data protection must also include mechanisms for enforcing access control policies, to support the privacy and security principles of least privilege and data minimization, where data is protected by limiting unnecessary
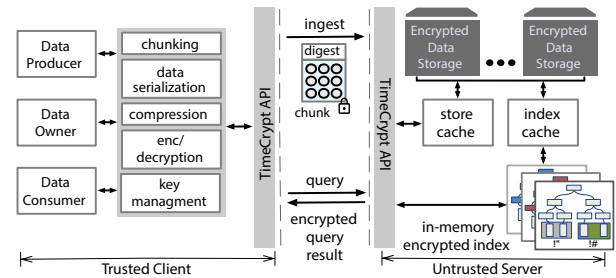

Figure 1: TimeCrypt's architecture.

exposure. State-of-the-art relational databases have security mechanisms designed for this purpose. The most adopted approach to support access control is based on views[2] and row-level access policies. However, specifying effective access control policies necessitates taking into consideration the semantics of data. Therefore, we investigated the major state-of-the-art time series databases [1, 4, 12, 42, 58, 65, 78] to understand the state of affairs in stream data access policies. We found that the only access policy restriction provided at the database interface, if any, is at the stream unit (i.e., grant or decline access to the entire stream). This binary protection level is however too coarse. This prompted the following question: What type of policies can offer the fine-grained protection that is required for selective and secure sharing of data streams? Stream data access control literature [24, 71] and time series applications designed for multi-user settings[5, 40] both recognize that policies which are expressed in time, resolution, and attributes are ideal for fine-grained access restrictions on streams. Examples of such policies can be a user choosing to simultaneously share hourly averages of their measured heart rate with their doctor and per-minute averages with their trainer but only for the duration of their workout session. Similarly, a datacenter operator might share resource utilization levels with a tenant but only for the duration of her job. Our goal is to translate these stream-specific sharing semantics into a cryptographically enforceable access control mechanism.

2.4 Threat Model

Our goal is to maintain the confidentiality and integrity of computations running on a cloud infrastructure that is potentially subject to an adversary that can read and tamper with data and manipulate query execution. In order to support sharing, we require a public-key infrastructure, such that entities can be identified and that a private/public key-pair can be associated with them. TimeCrypt provides the following guarantees in this setting:

Confidentiality. Data is encrypted using semantically secure encryption before it leaves the client device. Since decryption keys are never disclosed to the cloud provider, data confidentiality is guaranteed even in the case of a system compromise

---

[2]A view; also referred to as virtual table, is a dynamic window of a subset of the rows and columns in a database.

or malicious provider. Note that we do not employ property-revealing encryption, avoiding their inherent information leakage issues [55]. Our cryptographic access control mechanism ensures that data consumers can only query and access data according to the access policies defined by the data owner.

Integrity. TimeCrypt's integrity protection guarantees that, if a query completes, its output is equivalent to a correct execution on a trusted platform. Therefore, a malicious server cannot affect the computation, except by denying service. Note that even in case an integrity key for a stream is leaked, confidentiality remains intact.

Access Patterns. Similar to previous work [60, 63, 74, 81], TimeCrypt is non-oblivious, i.e., it does not protect against access pattern-based inferences in a trade-off for performance and scalability. Therefore, an adversary can learn which data the consumers are authorized to access by observing access patterns. TimeCrypt could be complemented with Oblivious RAM approaches [67] to hide these access patterns.

Access Control Collusion. Resolution based sharing in combination with interval sharing is not collusion resistant, even when considering a plaintext system. For example, any entity with access to aggregation over the intervals $[t_0, t_2)$ and $[t_1, t_2)$, can trivially derive the aggregation $[t_0, t_1)$ over the overlapped range by computing the difference. Hence, clients must be careful when sharing different resolutions over overlapping intervals. Furthermore, TimeCrypt comes with a trade-off between performance and collusion resistance when sharing non-continuous intervals. In the default mode, an adversary with access to two non-continuous intervals can compute the aggregation between the two intervals. For cases where this poses a privacy risk to applications, TimeCrypt provides a mechanism to prevent such collusion (§3.1) at the cost of increased decryption time.

A full security analysis with formal definitions and proofs can be found in the appendix of the extended version of the paper [23].

2.5 TimeCrypt Approach

TimeCrypt is a new encrypted time series database design that meets the scalability and low-latency requirements associated with time series workloads. We propose a new approach for data stream encryption that supports processing over encrypted data streams, computation integrity, and powerful access control within a unified scheme.

Data Abstraction. TimeCrypt stores data points in a stream as time-ordered chunks of predefined time intervals, i.e., $[t_i, t_{i+1})$ with a fixed interval size $\Delta = t_{i+1} - t_i$. Each data chunk also includes an encrypted digest that consists of statistical summaries about the underlying data. The digest enables TimeCrypt to compute statistical queries over time ranges efficiently, as we discuss next. At the client, the chunks are encrypted with standard symmetric encryption while the digests are encrypted with HEAC.

Aggregatable Digests. As HEAC is additively homomorphic[3], it supports secure aggregation of ciphertexts. However, to support queries beyond sum, we leverage *aggregatable encoding techniques* that exist in literature to support sophisticated statistical and analytical queries over encrypted data. At a high level, we introduce a per-chunk digest, which holds a vector of encoded values $\{x_0, ..., x_n\}$ that are encrypted with HEAC. To process queries, the server computes the aggregate function on the encrypted encodings across different digests. With this, we can support statistical queries that are inherently aggregation-based (e.g., sum, mean) or can be transformed to be aggregation-based (e.g., min/max, regression) (§4).

Encryption and Access Control. A key aspect of our scheme is tied to the observation that time series data streams are continuous. Consequently, to enable encrypted data processing that natively supports access control, we model data streams as a series of time segments, where each segment is encrypted with a different encryption key. We introduce a time-encoded keystream that maps keys to segments of the data stream, such that when a user restricts access to the data stream, only the corresponding range in the keystream is shared with the data consumer (§3.1). Based on the access policy, a data consumer is provided with the necessary decryption keys via an access tokens. Access tokens are encrypted with the data consumer's public key (hybrid encryption) and stored at the server. To enable sharing without enumerating all the keys and to support a *succinct key state*, we derive keys from a hierarchical tree key-derivation construction (§3.3). We also introduce a technique to support restricting access to a particular resolution level (§3.3.2), e.g., aggregated values at 10-minute resolution.

3 Encryption in TimeCrypt

In this section, we introduce the cryptographic components of TimeCrypt and present HEAC in more detail. HEAC, in essence is based on a symmetric homomorphic encryption [27]. However, we improve its performance by a factor of 2x for time series workloads by mapping keys to time and optimizing it for in-range ciphertext aggregations. Furthermore, we extend it to support fine-grained cryptographic access control capabilities tailored to time series data. Finally, we ensure computation integrity on encrypted data via Homomorphic Message Authentication Codes.

3.1 Symmetric Homomorphic Encryption.

We encrypt an integer m_i from the message space $[0, M-1]$ as $c_i = Enc_{k_i}(m_i) = m_i + k_i \bmod M$, with key $k_i \in [0, M-1]$. Given k_i, one can decrypt c_i as $Dec_{k_i}(c_i) = c_i - k_i \bmod M = m_i$. This scheme is semantically secure when the keys are pseudorandom and no key is reused [27].

---

[3]An additive homomorphic encryption scheme supports additions on ciphertexts, such that $decrypt(C_1 \oplus C_2) = decrypt(C_1) + decrypt(C_2)$.

Given the aggregated secret keys, one can decrypt the aggregated ciphertexts:

$$\sum_{i=0}^{n} m_i = Dec_{\sum_{i=0}^{n} k_i}(\sum_{i=0}^{n} c_i) = \sum_{i=0}^{n} c_i - \sum_{i=0}^{n} k_i \bmod M \quad (1)$$

We set M to 2^{64}, to support all integer sizes, without leaking any information about their original size.

Key Canceling. In the above scheme, the local computation to aggregate keys is linear in the number of aggregated ciphertexts, forcing the client to perform the same amount of computations as the server. We reduce this linear overhead to a constant, by leveraging the fact that time series data is generally aggregated in-range (i.e., over a contiguous range in time) as discussed in §2.1. We can therefore employ *key canceling* [6, 19, 31, 60]. This technique will also be relevant later, when we discuss integrity and access control (§3.2, §3.3). To enable this optimization, we choose the individual encryption keys such that the inner keys cancel each other out during aggregation. We do this by replacing the individual key k_i with a composite key that links subsequent messages:

$$Enc_{k'_i}(m_i) = m_i + k'_i \bmod M, \text{ with } k'_i = k_i - k_{i+1} \quad (2)$$

For decryption of an in-range aggregated ciphertext (Eq. 1), we now require only the two boundary keys:

$$\sum_{i=0}^{n} k'_i = (k_0 - \cancel{k_1}) + (\cancel{k_1} - \cancel{k_2}) \ldots (\cancel{k_n} - k_{n+1}) \quad (3)$$

With key canceling, the decryption time in TimeCrypt is independent of the number of in-range aggregated ciphertexts. This scheme remains semantically secure [23, 27, 31]; an attacker without access to the keys cannot exploit the canceling property. However, when given access to keys for two non-continuous intervals, an adversary could learn aggregates about the skipped time between the two intervals. For example, when given access to $k_0, \ldots, k_5$ and $k_{10}, \ldots, k_{15}$, they could compute $\sum_{i=5}^{10} m_i \bmod M$, given k_5 and k_{10}. The ramifications of this issue arise when users share adjacent intervals in the same stream with small gaps. TimeCrypt provides a hybrid key-canceling mechanism that limits this leakage in a trade-off for longer decryption times. We split the keys into epochs by replacing some k_i with non-canceling skip-keys k'_i, k''_i in $k_{i-1} - k_i$ and $k_i - k_{i+1}$, respectively. With this, we can share one interval per epoch without leakage. This increases the cost of aggregations over the epoch borders by two key derivations and one addition.

Time-Encoded Keystream. In TimeCrypt, access permissions are expressed with temporal ranges, e.g., Sep-14-15:00 till Sep-17-06:00 2019. Internally, TimeCrypt *chunks* data into fixed time segments of size Δ, which can be set per stream (e.g., 10 s intervals). In addition to the raw data points, each chunk is augmented with digests that are used for statistical query processing. Each chunk is encrypted with a fresh key from the keystream, indexed by the time window of the chunk. Assuming the data stream starts at timestamp t_0, the chunk digest m_i for the interval from t_i to t_{i+1} is encrypted as $c_i = Enc_{k_i - k_{i+1}}(m_i)$. By mapping keys to temporal ranges, a time range implicitly determines the position of the used key in the keystream. As a result, we sidestep the need to store identifiers of the keys along with the ciphertexts and avoid ciphertext expansion.

3.2 Integrity

Homomorphic encryption schemes are by design malleable, and therefore susceptible to ciphertext manipulation. In our setting, a dishonest server could try to drop, duplicate, or manipulate ciphertexts, resulting in incorrect query outputs. Incentives for deviations from the protocol could be as simple as trying to preserve resources by reducing the complexity of queries [72]. Beyond malicious behavior, integrity checks help to prevent faulty executions (e.g., data corruption, hardware faults, or misconfigurations). Ensuring computation integrity is essential, but is rarely considered in existing encrypted databases. Computation integrity can be achieved by requiring the server to provide a proof that the encrypted result was computed using the targeted data and function. Along this line, we introduce a verification protocol that allows the server to validate the output of in-range aggregations over ciphertexts with a succinct tag that can be verified in constant time at the client. To generate the proof, we use homomorphic Message Authentication Codes (HoMAC) [28]. While HoMACs have been introduced as cryptographic building blocks in the literature, existing solutions do not achieve integrity while maintaining scalability.

HoMAC. Conventional Message Authentication Codes (MACs) are small tags generated for each ciphertext which later ensure the authenticity and integrity of the ciphertext. *HoMACs* [9, 28] are conceptually similar to MACs, but additionally allow the server to perform computations like aggregations over the ciphertexts, and to produce new tags that authenticate the outputs of the computation. More precisely, the client generates a HoMAC tag σ for each ciphertext c and uploads (c, σ), where σ is defined as follows:

$$\sigma = HoMAC_s(c) = (s - c)/Z \mod p \quad (4)$$

where s is a per-ciphertext key, Z the HoMAC key, and p a prime number. The server computes aggregations on both the ciphertext and *HoMAC* tags $\sum_{i=0}^{n-1}(c_i, \sigma_i) = (c_{res}, \sigma_{res})$. The resulting tag σ_{res} authenticates and verifies that the output c_{res} corresponds to that specific aggregation. A client in possession of the *HoMAC* key material can verify the result by checking that the received σ_{res} tag matches the ciphertext c_{res}:

$$\sum_{i=0}^{n-1} s_i \overset{?}{=} c_{res} + \sigma_{res} Z \mod p \quad (5)$$

HoMACs are interesting for our use-case, since their symmetric nature makes them appealing to integrate with HEAC.

In contrast to authenticated data structures [46, 84], which can be used for outsourced computation verification, HoMAC tags do not need to be updated when new data is inserted. However, without further optimization, their verification overhead prevents their use in our setting.

Integrity Protocol. While HoMACs provide the desired integrity guarantees, they suffer from a verification overhead that is linear in the number of records in the aggregation query. Therefore, we apply a similar key canceling technique as already discussed above in the context of encryption: We define a *HoMAC* keystream $\{s_0, s_1, s_2, ...\}$ and, for each ciphertext c_i, the client computes the *HoMAC* tag σ_i as follows:

$$HoMAC_{s_i'}(c_i) = (s_i' - c_i)/Z = (s_i - s_{i+1} - c_i)/Z \mod p \quad (6)$$

Setting $s_i' = s_i - s_{i+1}$ enables a constant time verification at the client side regardless of the input size, since only the two outer keys are required:

$$\sum_{i=0}^{n-1} s_i' = s_0 - s_n \stackrel{?}{=} c_{res} + \sigma_{res} Z \mod p \quad (7)$$

Using the key canceling concept in both encryption and integrity is a key enabler for our efficient cryptographic access control (§3.3). Since verification of aggregation results does not require access to the individual messages that were aggregated, our integrity protocol also integrates well with the resolution-based access control (§3.3.2).

HoMAC Security. For an attacker, it is computationally infeasible to generate a forged ciphertext and a tag which pass the verification. Note that we use different HoMAC key streams not just per-stream, but also per type of digest, i.e., target function. Therefore, the server cannot substitute a digest aggregation with another. In the case of key leakage, a party with access to the HoMAC key Z would be able to forge tags, but data confidentiality always remains intact. For a complete security treatment of HoMAC we refer to [28, 31] and the extended paper [23].

3.3 Cryptographic Access Control

The symmetric homomorphic encryption and HoMAC both require a pseudorandom keystream with one key for each message. The conventional approach to efficiently generating such keystreams would be to leverage a pseudorandom function with an initially exchanged secret key. This allows handling a large number of keys with one secret. However, with this approach, one could only share the entire data stream (i.e., all-or-none or in other words no fine-grained access control). Instead, we want to allow efficient sharing of arbitrary intervals, and want to allow users to restrict access to lower-resolution data, e.g., hourly or daily summaries. To realize this granular access control and to allow data owners to cryptographically enforce the scope of access to their data, we design a novel key derivation construction.
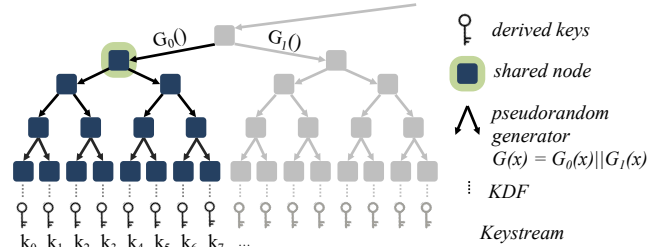

Figure 2: TimeCrypt's key derivation tree (leafs form a keystream).

3.3.1 Key Derivation Trees

Our key derivation is based on *key derivation trees*, i.e. balanced binary trees where each node contains a unique pseudorandom string. The leaf nodes represent the inputs to a key derivation function (KDF) to compute the keystream $\{k_0, k_1, k_2, ..., k_{2^h-1}\}$ as depicted in Fig. 2. The key derivation tree is built top-down from a secret random seed as the root. The child nodes are generated with a pseudorandom generator (PRG) that takes the parent string as the input. Our PRG consists of $G_0(x)$ for the left-hand child and $G_1(x)$ for the right-hand child, where x is the parent node. This procedure is applied recursively until the desired depth h in the tree is reached. We select a large h such that the keystream is virtually infinite, especially when considering that high-frequency streams will be chunked into e.g., one chunk per second. The pseudorandom generator can be realized from hash functions $G_0(x) = H(0||x), G_1(x) = H(1||x)$ with x as the key.

Access Token. The key derivation tree allows us to share segments of the keystream efficiently. Instead of sharing the segment key-by-key, the client shares a few inner tree nodes, combined into an *access token*. For instance in Fig. 2's toy example, a data owner grants access to the stream from t_0 to t_7, and the corresponding key segment $\{k_0, ..., k_7\}$ is shared using a single node. In practice, a single node in the tree can be used to share thousands of keys. Note that given a node it is computationally not feasible (i.e., due to one-way property of PRGs) to compute the parent, sibling, or any of the ancestor nodes. Hence, a data consumer cannot compute any keys outside the segment they are granted access to.

Token Distribution. Once the data owner specifies an access policy for a data consumer, the TimeCrypt client generates an access token which encapsulates the inner nodes of the tree needed to derive the corresponding shared keystream segment specified in the access policy. We use the same key derivation tree for the encryption and HoMAC keystreams, but with a different KDF[4]. The token also contains encoded information about the subtree height and key identifier offset. TimeCrypt then encrypts the tokens with the data consumer's public key (i.e., hybrid encryption) and stores it at the server, such that the data consumer can fetch it to gain access to the keying

---

[4]Each leaf node of the primary key derivation tree is used to produce cryptographic keys needed for its corresponding chunk. Namely, keys for each element in the digest (i.e., query type), chunk, and HoMAC. Hence, we use different KDFs with the same node.
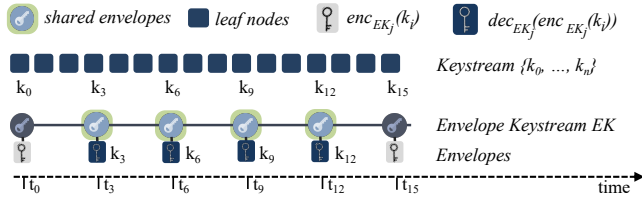
Figure 3: Envelope encryption for resolution-based access, showing envelopes required to share $[t_3, t_{12}]$ at a resolution of 3Δ.

material required to decrypt the data or query results. Note that TimeCrypt's key distribution is pluggable and we can employ alternative solutions. For instance, we can encrypt the token with attribute based encryption [83] to share tokens based on attributes (e.g., month as a key attribute).

3.3.2 Resolution-based Access Restriction

We now discuss how TimeCrypt provides crypto-enforced access control over the *resolution* at which data can be queried; i.e., the data owner not only restricts access to a time range per data consumer but also defines the temporal granularity (e.g., per minute) at which they can retrieve or query data.
Resolution Levels. In TimeCrypt, the highest resolution for queries and access control is defined by the chunk size Δ. Whenever we aggregate over an interval, we reduce the data resolution. For example, with one second chunks, an aggregation over 60 chunks results in a per-minute resolution. We can exploit the fact that keys cancel out during in-range aggregations, as described in §3.1, to cryptographically restrict access to lower resolution levels. In general, a ciphertext generated through an in-range aggregation over the time period $[t_i, t_j]$ has the form:

$$\sum_{x=i}^{j-1} c_x = \sum_{x=i}^{j-1} m_x + k_i - k_j \qquad (8)$$

where the inner keys are canceled out. Hence, given access to just the boundary keys k_i and k_j, one can decrypt the aggregation, but none of the individual ciphertexts. Resolution levels must be multiples of the chunk size Δ and the segments at a given level must not overlap. Otherwise, data consumers could compute the difference of two aggregates overlapping by e.g., one chunk, allowing them to learn the data for that chunk which would violate the resolution-based access policy. For example, if the data owner wants to restrict access to a 3-fold resolution of the chunk size, the data owner would share only $\{k_0, k_3, k_6, ...\}$ with the data consumer. The data consumer can then decrypt the aggregated ciphertexts at the 3-fold (i.e., $3 \cdot \Delta$) or lower resolutions, but cannot access higher resolutions since the inner keys are missing.
Envelopes. While a data owner could share the boundary keys required for resolution-based access directly, this is not efficient since the number of keys necessary is linear in the length of the shared interval. Instead, the data producer stores the required boundary keys for a stream on the

server, protected by another layer of encryption, the *envelope*. The keys used for the envelope encryption are derived from a new tree-based keystream $\{\bar{k}_0, \bar{k}_1, \bar{k}_2, \ldots\}$. For each resolution level, we use a different keystream for the envelope encryption. For example, if a data owner wants to make a per-minute resolution available for a stream with 20 s data chunks, the data owner encrypts the boundary keys $\{k_0, k_3, k_6, \ldots\}$ with the envelope keystream, and stores $\{enc_{\bar{k}_1}(k_0), enc_{\bar{k}_2}(k_3), enc_{\bar{k}_3}(k_9), \ldots\}$ on the server, as shown in Fig. 3.

Sharing a stream at a lower resolution is then again a matter of sharing a single access token, with the difference that the token now contains nodes of the key derivation tree for the envelope keystream, rather than for the original encryption keystream. A lower-resolution query returns, in addition to the encrypted result, two envelopes containing the two boundary keys required to decrypt the aggregated ciphertext. The overhead of resolution-based access control is similar to access control without resolution restrictions (§3.3), i.e., an access token consist of at most $O(log(n))$ nodes from the key derivation tree.
Dynamic Resolution Levels. In TimeCrypt, a user does not need to decide a priori on a fixed resolution for data consumers and can dynamically at any point in time define a new resolution. E.g., Alice can share her health data with a physician at minute-level (high-resolution) during physiotherapy from Jan-to-Feb, and from March reduce the resolution to hourly (low-resolution). The physician only sees high-resolution data for Jan-Feb and only hourly-data from March onwards.

3.4 Access Control Extensibility

Beyond temporal and resolution-based access policies, our construction also lends itself to enabling privacy policies on encrypted data, as combining ciphertexts from multiple users creates valid ciphertexts under a new *virtual aggregate key*. In the context of private operations, privacy policies permit a data consumer (e.g., analyst) to only run cross-stream aggregate queries, without having access to individual data streams. Similar to data access policies, privacy policies in our system are enforceable via encryption. As a concrete example, a user might want to allow a research lab to query her data but only if aggregated with a fixed set of n users, to preserve her individual privacy. Ensuring that a data consumer can only decrypt aggregates across a set of users can be realized by ensuring that she only has access to the *virtual aggregate key* (i.e., the data consumer never sees the keys for a particular user's stream in isolation). For instance, if a service is authorized to access an aggregate query over n encrypted messages from different users, then sharing only the *virtual aggregate key* $\sum_{i=1}^{i=n} k_i$ will ensure that the analyst can only decrypt the aggregated result. Therefore, we need a way to compute the *virtual aggregate key* without exposing the individual keys k_i of each user to any of the involved parties; the storage provider, authorized data consumer, or other users.

Function	Description
(1) `CreateStream(uuid, [config])`	Create a new stream, config defines parameters, e.g., chunk interval, operators.
(2) `DeleteStream(uuid)`	Delete specified stream with all associated data.
(3) `RollupStream(uuid, res, [T_s, T_e])`	Rollup an existing stream or a segment of it to the specified resolution.
(4) `InsertRecord(uuid, [t, val])`	Serialize data points in a chunk and append to the end of the stream.
(5) `GetRange(uuid, T_s, T_e)`	Retrieve all data records within the specified time interval.
(6) `GetStatRange([uuid], T_s, T_e, resolution, [operators])`	Retrieve statistics for the given time interval and resolution, default [sum, count, mean, var, freq].
(7) `DeleteRange(uuid, start, end)`	Delete specified segment of the stream, while maintaining per-chunk digest.
(8) `GrantViewAccess(viewid, [princ-id])`	Grant access to an existing View.
(9) `CreateView(viewid, [policy])`	Create a View with the given policy in JSON format.
(10) `CheckView(viewid, princ-id)`	Retrieve a View token.

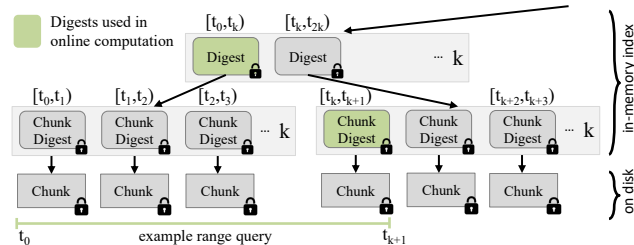Table 1: TimeCrypt's basic API.

Figure 4: A statistical index for time series data with a k-ary time-partitioned aggregation tree. The pre-computed encrypted index allows for fast response times for statistical queries.

This can be accomplished by a secure aggregation protocol [6, 19] between the involved users and the analyst. The inputs to the protocol are the users' individual keys k_i and the output is the blinded contributions towards the *virtual aggregate key*. Queries across streams can be performed efficiently on the server, and the analyst can only decrypt the final result via a *virtual aggregate key*. In §6, we discuss a private crowdsourcing application atop of TimeCrypt that uses this technique.

4 Fast Analytics and Processing

To meet the requirements of time series databases, TimeCrypt must handle massive amounts of data, yet at the same time be able to serve queries with low latency. We address this challenge by introducing efficient client-side serialization/encryption and efficient encrypted indices on the server.

Client-side Data Serialization. The client serializes and encrypts data chunks containing the raw data, and digests. The content of a digest is set per stream based on the supported queries. The *default* query configuration of TimeCrypt supports *sum*, *count*, and *mean*. Other query types such as *variance*, *standard deviation*, *histogram*, bucket *min/max*, approximated *quantiles*, *trend detection*, and limited *filter* queries, can be enabled.

Server-side In-memory Encrypted Index. TimeCrypt's server maintains an in-memory encrypted index based on a time-partitioned aggregation tree over encrypted data. This is a key building-block that enables us to serve low-latency analytics on large encrypted data streams and enables efficient data retention. The index structure is a k-ary tree, where each internal node (digest) holds k statistical summaries of the subtree below it. The tree leaves store the chunk digests encrypted

with HEAC at the client and represent the highest resolution data summaries (Fig. 4). On the arrival of a new chunk digest, the server inserts it as a leaf node, and updates statistical summaries of the parent nodes's by performing an encrypted aggregation. Any operation that can be expressed as an aggregation of the intermediate results from the child subtrees can be included in the summaries (see §4). Time series workloads are in-order and append-only, therefore updating the tree is straightforward. The encrypted index enables TimeCrypt to significantly decrease the response time for statistical queries, as the server avoids expensive serial scans. When executing a statistical range query over a time interval, the server traverses the tree and selects only the digests required to cover this interval, as illustrated in Fig. 4.

Statistical Queries. So far, we have developed the means to evaluate aggregates over ciphertexts, now we briefly[5] discuss how we combine aggregation with known encoding techniques [32, 47, 68] to allow TimeCrypt to compute more sophisticated statistics over ciphertexts. At a high level, each per-chunk digest holds a vector of encoded values that are encrypted with HEAC. For example, this vector might include the encrypted *sum* and *count* of the data points in the chunk. From this, we can then also calculate the *mean*. To compute quadratic functions, e.g., *var* and *stdev*, the vector includes the *sum of squares* of the points in the chunk. We can also include the *frequency count* of data points in the chunk, which yields valuable information to compute several statistical functions, such as *min, max, top N, bottom N, histograms, and quantiles.* For frequency counts, we use a vector $[c_{v_1}, .., c_{v_n}]$, where each element in the vector c_{v_i} tracks the count of data points with value v_i. This works well for small n, which is often the case for (discrete) time series data. For larger ranges of values, we approximate the frequency count, i.e., each c_{v_i} tracks the count of a small range (*bin*) around v_i [32].

Advanced Analytics. In principle, any operations with aggregatable transformations can be supported in TimeCrypt, including a variety of sketch algorithms [52]. In addition, we can support many forms of machine learning, e.g., via aggregation-based encodings that allow private training of linear models [32, 47, 68]. These types of analytics are often employed in time series data to understand and detect runtime

---

[5]Due to space constraints, we keep the description here brief and refer to [32, 47, 68] for detailed description.

anomalies, trends, and patterns. We show how such analytics can be realized in TimeCrypt, using the example of *private trend detection*, i.e., identifying a general tendency over a defined time interval. It allows users to estimate the magnitude of a trend and is a highly related task to event detection (e.g., runtime anomalies). Linear regression using least-squares is a simple yet powerful method for trend detection [15]. To compute a linear regression model over a stream, the per chunk digest is defined as $(\sum_i x_i, \sum_i t_i x_i)$ for $i \in [0, n)$. This way the expensive aggregations are done at the server. Such learning on summarized data also delivers privacy gains, as the raw data is not exposed in the training phase. In §6, we discuss the performance aspects of implementing such applications atop TimeCrypt.

Filter Queries. TimeCrypt supports filter queries with predefined predicates. One can define digest encodings that contain statistics over the values of the underlying chunk filtered with a predicate P (e.g., the sum of all values larger than 10). In the query phase, the filtered digests are used to compute statistics over the values matching the predicate P.

Time-Decayed Data Processing. As time series data ages, it is often aggregated into lower resolutions for long-term retention of historical data, while high-resolution data is aged-out. Typical strategies are based on compact summaries through aggregates [7, 43, 80]. TimeCrypt natively supports these approaches: as our index maintains aggregated summaries of the raw data, we can selectively delete aged-out raw data and prune lower nodes in the index. For example, we implement a retention policy based on the time-decayed merge algorithm [7] which keeps the data store compact (logarithmic in the input size) by dynamically re-compacting older data as new data arrives.

5 Prototype

API. TimeCrypt is realized as a service which exposes an interface similar to conventional time series stores [7, 12, 50]; applications can insert encrypted data, retrieve encrypted data by specifying an arbitrary time range, and process statistical queries over arbitrary ranges of encrypted data, as summarized in Table 1. In TimeCrypt, each stream is identified by a unique UUID and associated *stream metadata*, e.g., hostname, data type, sensor ID, location. Each stream has one writer (i.e., data producer) and one or multiple readers (i.e., consumers). A data owner can grant and specify access polices to consumers.

Granting Access to Stream Views. Data owners can manage access to their stream resources with the View API. *View*s define what a data consumer can access within the scope of the *View*. *View*s are set in JSON format, containing a unique identifier and a list of per stream access policies. In the current version, an owner can define the time range and the granularity that is accessible per stream, as for example: *"viewid": 2999, "streams": [{"uuid": [9,10], "from": "1546315200", "to": "1546315800", "granularity": "60s" }]*. This *View* defines an access scoped to stream 9 and 10 in the specified time window with a minute granularity. After the user defines a policy, the API assembles the access token with the necessary inner nodes of the key construction for the specified *View* (§3.3). The client library then derives a *View* key, encrypts the token along with the JSON description using AES-GCM and uploads it to the server. To give data consumers access to the *View*, the client invokes the *GrantViewAccess* command, which encrypts the *View* key with the respective consumer's public keys. The authorized consumers can download the tokens for the given *View* and can query the streams in the defined scope by the access policy. Though access policies are enforced by encryption, the intricacies of the key management are insulated from users in our design.

Reference Implementation. TimeCrypt's prototype is implemented in Java and consists of 6k SLOC with additional 4k SLOC for the applications and benchmark code. We used *Netty* [44] for network communication. TimeCrypt's server and client communicate over Google's *protobuffers* [37] protocol. The current prototype uses Cassandra [26] as the storage backend. The encrypted index is augmented with the in-memory cache *caffeine* [51] to speed up index node access. For the implementation of the cryptographic schemes, we used the Java security provider and a native C implementation of AES-NI. We compare the encrypted index performance with HEAC against alternative private aggregation schemes. We implemented three variants of the encrypted index based on Paillier [77] (Java *BigIntegers*), EC-ElGamal (OpenSSL [57]), and ASHE (we implemented it as described in [60]). Our code is available online.

6 Evaluation

In this section, we evaluate TimeCrypt's practicality. Our evaluation answers three core questions: *(1)* Can TimeCrypt meet the performance requirements of time series applications? *(2)* What are the performance gains of HEAC compared to alternatives? — HEAC supports access control and secure computation simultaneously; both aspects have traditionally been addressed with different schemes, consequently we examine alternatives independently in our evaluation. *(3)* Can TimeCrypt run compelling real-world applications?

Setup. Our experiments are conducted in Amazon AWS, on M5 instances equipped with a 2.5 GHz CPU running Ubuntu (16.04 LTS). TimeCrypt's server runs on an m5.2xlarge instance with 8 virtual processor cores (vcores) and 32 GB of RAM and a Cassandra node runs on an m5.xlarge instance with 4 vcores and 16 GB of RAM. The clients are simulated on several m5.xlarge instances. The client and server are located in the same data center network, with up to 10 Gbps bandwidth. In the microbenchmark, we quantify the overheads of encryption and decryption on end devices. We consider resource-constrained IoT devices; this class of devices is a major source of sensitive time series data. We use IoT OpenMotes (32-bit ARM M3 SoC 32 MHz) and a MacBook Pro 2.8 GHz Intel Core i7, with 16 GB RAM.

System	Micro	Index - Size	Average Ingest Time			Average Query Time (worst-case)		
	ADD	1M	1k	1M	100M	1k	1M	100M
TimeCrypt	1ns	8.1MB (1x)	10μs (1.7x)	16μs (1.3x)	22μs (1.3x)	21μs (1.6x)	46μs (1.3x)	50μs (1.1x)
TimeCrypt+	3ns	24.3MB (3x)	16μs (2.6x)	35μs (2.9x)	39μs (2.3x)	38μs (2.9x)	87μs (2.4x)	109μs (2.4x)
Plaintext	1ns	8.1MB (1x)	6μs (1x)	12μs (1x)	17μs (1x)	13μs (1x)	36μs (1x)	45μs (1x)

Table 2: Overview of evaluation results on the cloud, with 128-bit security, except for plaintext. The largest index size with 100M chunks, represents 50 billion data points in our health app.
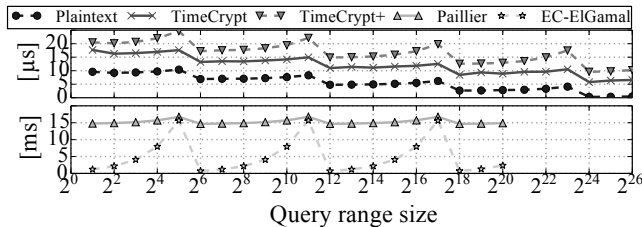

Figure 5: Aggregate queries over varying time ranges (i.e., query range size). Aggregating the entire index corresponds to retrieving the encrypted root.

We quantify the overhead of **TimeCrypt** (confidentiality) and **TimeCrypt+** (confidentiality plus query verification), and compare it to *(i)* operating on **plaintext** as the baseline, and *(ii)* **prior work** where we consider alternative encryption schemes for encrypting the digest, i.e., *Paillier* (used in CryptDB [63]), *EC-ElGamal* (used in Pilatus [74]) and *ASHE* (used in Seabed [60]). For access control, we compare to a strawman solution and a construction of KP-ABE (used in Sieve [83]), that we use to realize temporal access control similar to that supported by HEAC. Unless noted otherwise, we use 128-bit security [13], i.e., 3072-bit keys for Paillier and 256-bit elliptic curves for EC-ElGamal (i.e., prime256v1). For the microbenchmarks, we use synthetic large data that resembles the mhealth application (§6.4) dataset.

6.1 Encrypted Data Processing Performance

We now discuss the evaluation results of different aspects of the encrypted index, as summarized in Table 2. In the microbenchmark, the index supports one statistical operation (i.e., sum) for isolated overhead quantification, whereas in the E2E benchmark the index supports all our default queries. In all experiments, we instantiate 64-ary index trees and a keystream with one billion keys via the key derivation tree.

Index Size Expansion. To improve query efficiency, in-memory time series databases aggressively seek to reduce storage footprint, to support a model where almost all recent data can be stored in memory. When considering encryption for time series data, the degree of ciphertext expansion has a direct impact on the encrypted index storage footprint, hence impacting query efficiency. TimeCrypt has no ciphertext expansion for 64-bit values, TimeCrypt+ introduces a 128-bit expansion due to the HoMAC tag. The encryption schemes in prior work [63, 75, 81] exhibit large ciphertext expansion, e.g., for one million chunks we experience 96x index

size expansion with Paillier. Hence, limiting the performance gains of in-memory processing and impacting query latency. ASHE [60] uses an encoding where the expansion depends on the order of aggregation. With in-range aggregation this amounts to 12.5% higher expansion compared to TimeCrypt.
Ingest Time. On each ingest, i.e., insertion of a leaf node, statistical aggregates of ancestor nodes are updated. In Time-Crypt, additions are as efficient as in *plaintext*. Hence, the average ingest time increases slightly due to the encryption cost; 1.3x for the large index. With verification the average ingest time increases by 3.2x due to the HoMAC overhead.
Query Performance. Fig. 5 shows the performance of the index for statistical range queries of different lengths, i.e., $[0, 2^x]$ with $x \in [0..26]$. As the length of queries increases fewer tree levels are traversed, which results in fewer cache fetches and lower computation time, e.g., the index depth of five is observable in Fig. 5. For plaintext and TimeCrypt the resulting pattern is similar due to the low cost of additions, while for TimeCrypt the decryption overhead is visible. Queries with non-power-of-k ranges require an index *drill down* on either end of the range. This increases the computation time logarithmic, $O(2(k\text{-}1)log_k(n))$ for a worst-case alignment, and not linear to the *n* stored chunks.
Comparison to Alternatives. In Fig. 6, we show HEAC's performance gains relative to the encryption schemes used in the other encrypted systems. For this experiment we launch an ingest/query workload, with one machine and 100 threads, where each thread constantly performs four statistical queries after each chunk ingest. The plaintext setting reaches a throughput of 5.77M records/s for ingest and 46.1k ops/s for statistical queries, as shown in Fig. 6a-b. TimeCrypt demonstrates an outstanding throughput for both ingest and statistical queries with only 2.9% slowdown compared to plaintext. With verification (TimeCrypt+), the slowdown increases to 7.8% due to the larger index size and HoMAC computations. TimeCrypt is by a factor of 2x, 20x, and 52x faster than ASHE, EC-ElGamal, and Paillier, respectively. Despite ASHE's lower encryption and decryption cost, the system throughput is by 2x lower due to the higher aggregation costs on the server. This is due to ASHE's key-encoding updates, which TimeCrypt eliminates with the time-to-key mapping. Fig. 6c-d shows the respective observed query latency. The impact of a small index cache (1 MB) is distinct, but similar for both plaintext and TimeCrypt, due to higher cache misses.
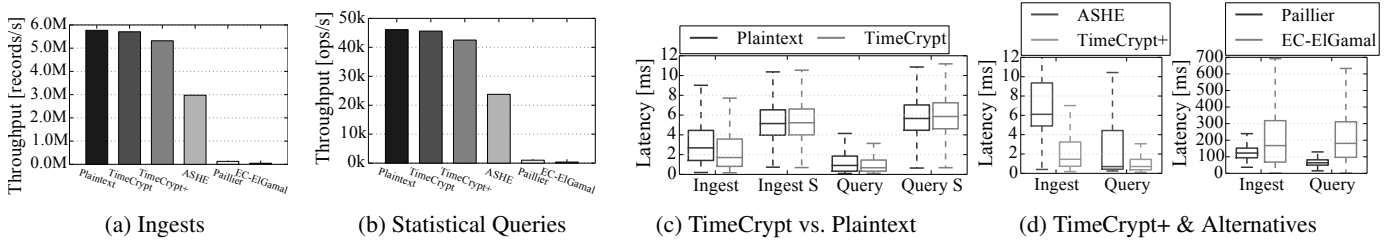
Figure 6: Latency and throughput for ingest and statistical queries for TimeCrypt with HEAC vs. EC-ElGamal, Paillier, & ASHE, and operating on plaintext indices. Heavy load experiment with a read-write ratio of 4 to 1, and also with extremely small (S) index cache (1 MB). The AWS load generator creates 1200 streams with 100 clients, corresponding to 48579 streams in our health app (Δ:10s, 50Hz data rate).

	HEAC		ASHE	Paillier
	[Enc/Dec]	[HoMAC]	[Enc/Dec]	[Enc/Dec]
IoT	1.08ms	20μs	0.3ms	1.59s / 1.62s
Laptop	5.1μs	0.2μs	1.5μs / 1.3μs	30ms / 15ms

Table 3: Performance of crypto operations with at least 80-bit security and 32-bit integers on IoT devices (OpenMote) vs. laptop (MacBook). TimeCrypt uses a key derivation tree with 2^{30} keys.

To compare how other encrypted systems perform while processing encrypted time series workloads, we run one aggregate query over one billion data records on CryptDB, Pilatus, Seabed, and TimeCrypt. Seabed requires seconds to process this query while CryptDB and Pilatus require minutes, whereas TimeCrypt can process such a query within a few milliseconds on a single machine.

6.2 Client Performance

Table 3 summarizes the enc/decryption and HoMAC costs of HEAC in TimeCrypt. TimeCrypt's cryptographic costs are dominated by the key derivation tree. Enc/decryption amount to 5.1 μs, which accounts for the time to compute the key. With HoMAC the clients incur 4% higher costs. To put this in prospective, this is three orders of magnitude faster than Paillier, EC-ElGamal, and ABE schemes with only few attributes. ASHE is faster in enc/decryption; the slight overhead in HEAC is due to the cost of deriving keys from our key derivation construction to support access control. Though overall, TimeCrypt is more performant in ingest and query performance due to its faster aggregations. The overhead of resolution-based access is defined by the access granularity. For instance, with 10 s chunk intervals and minute and hourly resolutions, the encryption cost increases by only 1% per day.

Low Power Devices. TimeCrypt is particularly compelling for battery-powered constrained devices used in the IoT and environmental sensing, where the power consumption of encryption is a serious challenge. Assuming one minute chunk intervals with TimeCrypt default queries, encryption consumes only 1.4% (400mJ) more battery per day on an Open-Mote device compared to sending data in the clear.

6.3 Access Control

In the following, we look at the performance and scalability of our encryption-based access control mechanism. The overhead can be quantified as the cost of key distribution, deriving HoMAC and encryption keys, and computing the resolution envelopes. To characterize the overhead, we consider an example scenario where a data owner has 1000 streams and shares a subset of each stream with a data consumer.

Naïve Key Management. TimeCrypt realizes access control by encrypting units of stream data with unique keys. Consequently, efficient key distribution is important for the scalability of this approach. In a naïve approach, data owner can compile all the keys associated with the specified access policy and distribute the keys encrypted individually to each principle. However, this leads to access tokens of size $O(n)$ where n is the number of keys (i.e., units of stream data included in the access policy). With our key derivation construction, we have a logarithmic worst-case complexity in the number of shared stream units $O(log(n))$.

Communication. An access token in TimeCrypt contains in the worst-case $2(\log(n) - 1)$ inner nodes of the tree key-derivation construction where n is the number of keys in the tree. This reduces the communication cost from a naïve approach from 50 GB to 1.28 MB, considering one year of data shared in our example scenario. With resolution-based access policies, the data consumer has to additionally download two envelopes per aggregation query (72 additional bytes).

Computation. Deriving the access token for all streams requires 145 ms. The decryption keys can be computed at a rate of 400k per second. With resolution-based access, the principal has to perform an additional decryption (for the envelope), which reduces the rate to 380k keys per second.

Storage. The storage cost can be broken down into two parts; key storage at the data consumer (1.28 MB), and resolution-related keying material at the server, which grows linearly with time (i.e., the envelopes). With a stream that consists of 10 s chunk intervals over one year with hour/day/month resolution support, the server stores 1.6 MB keying material (45.7k envelopes) per stream.

Comparison to an ABE-based Approach. Although, key-policy attribute-based encryption (KP-ABE) (used in Sieve [83]) is a powerful tool for access control, it comes
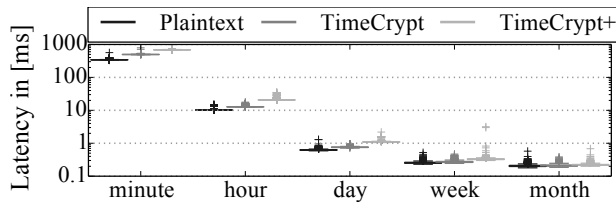
Figure 7: Latency in log-scale for statistical queries of one month data in our health app (121M records). The x-axis shows the granularity of the requested data from one minute to one month.

(a) DevOps Application (b) SmartEnergy Application

Figure 8: Applications: (a) DevOps trend detection queries on CPU utilization over different number of records. (b) Energy consumption queries for a day over multiple streams in a smart meter application.

with a relatively high computational cost, especially for low-power devices and when used to enable fine-grained polices as needed in time series data. Compared to KP-ABE (implementation from [3]), HEAC is three orders of magnitude more efficient for encryption/decryption. For an IoT device encrypting one chunk per minute, an ABE-based solution drains one order of magnitude more battery life compared to HEAC. Additionally, ABE does not support computation on encrypted data.
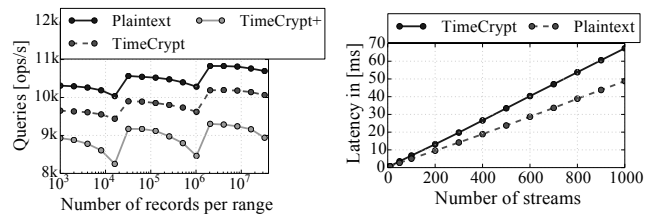
Interrupt Key Canceling. TimeCrypt can add epoch borders to reduce the risk of leakage from aggregating the skipped interval between two shared non-continuous intervals (§3.1). Each additional epoch border within the query range incurs an additional computational cost to decryption (i.e., one key derivation and two additions). For example, considering a weekly epoch and a daily epoch in a data stream, the decryption cost for a monthly aggregate result increases by a factor of 2.5x and 14.5x, respectively. However, even for fine-grained epochs (e.g., over 300 per range), the decryption latency remains well below 1 ms and would not impact user perception.

6.4 Applications

In this section, we evaluate the end-to-end overhead of Time-Crypt and its effectiveness in running complex, real-world applications. We developed four apps atop of TimeCrypt that represent different challenging requirements and workloads.

mHealth Views - Interactivity. We implemented an mHealth dashboard for the Biovotion health tracker [18]. The dashboard shows summary plots of the underlying data (i.e, windowed AVG). The data consists of 12 different metrics at 50 Hz from the Biovotion sensor over two weeks, which we stretch to one year worth of data. Fig. 7 shows the response time for aggregation plots of last month's data (121M records). We also consider the extreme case of plotting one-month data at minute granularity (403 MB plot), which induces an overhead of 1.45x (2.0x for TimeCrypt+) in latency compared to plaintext. With lower granularity, the overhead sharply decreases and reaches 1.06x (1.29x for TimeCrypt+).

DevOps Trend Detection - Complex Analytics. We developed a trend detection app for CPU utilization. We use a CPU monitoring dataset generated by the time series benchmark suite [79] with 10 metrics, 10s data rate, and per minute chunk size Δ over one year. The results of a two-dimensional linear regression model on different ranges of an encrypted CPU monitoring stream are shown in Fig. 8a. TimeCrypt matches the plaintext performance (0.75% slowdown).

Smart Energy Service - Access Control Scalability. We extended a smart meter application, where a service computes the aggregated energy consumption per day over households. Each smart meter uploads a chunk every 5s, but the service can only compute per day aggregates for each stream. We use the ECO dataset [48], which contains smart meter data sampled at 1 HZ rate and collected over 8 months. Fig. 8b shows the query latency for the aggregated energy consumption over up to 1000 streams. TimeCrypt's overhead is attributed to multi-stream processing and resolution-based access. The overhead stems from the linearly increasing decryption costs in the number of streams that are aggregated.

Crowdsourced mHealth - Privacy Policy Transformation. We enhance the mHealth app with a crowdsourcing feature which enables users to opt-in their data to be part of crowdsourcing for a targeted research project, as described in §3.4. For n users, the secure aggregation protocol [19] adds a communication overhead of n Diffie-Hellman key exchanges per user to create the envelopes. The envelope enc/decryption increases linearly (e.g., below 1 ms for 100 users).

7 Related Work

There is a large body of research on privacy-preserving systems, encrypted search, and secure outsourced computation. For brevity, we focus our discussion here on works that are closest to TimeCrypt.

Encrypted Databases. Fuller et al. [34] provide a comprehensive overview of the encrypted database landscape. We now discuss several works in this space that are analogues to TimeCrypt. CryptDB [63] and Monomi [81] augment relational databases with encrypted data processing capabilities, however, encryption schemes used in these systems are not efficient enough to support interactive queries on large data. Seabed [60] focuses on Spark-like batch processing workloads and resorts to symmetric partial-homomorphic encryption to enable interactive queries on big data but without the tight latency requirements of time series data. CryptDB, Monomi, and Seabed do not support cryptographic access control or verifiable computation, as the case with TimeCrypt.

ENKI [41] and Pilatus [74] support sharing and encrypted computations but they scale poorly with the number of principals and the size of data. Also, they do not support fine-grained policies. Bolt [40] is an encrypted data storage system for time series data that supports retrieval of encrypted chunks but does not support server-side computation on encrypted data or fine-grained sharing. BlindSeer [61] enables private boolean search queries over an encrypted database by building an index with Yao's garbled circuits and primarily targets private search over large data with no support for statistical queries. It integrates access control for search queries but requires two non-colluding parties. Another line of research considers building data processing systems in trusted execution environments [14, 64, 72], which can provide confidentiality and integrity of queries. In TimeCrypt, we do not require dedicated hardware and rely on cryptographic primitives to ensure confidentiality and integrity of computation.

Cryptography-based Access. Cryptographically enforced access control is explored by crypto-systems [35] such as identity-based encryption, attribute-based encryption (ABE), predicate encryption, and functional encryption. They enable complex access control to encrypted data. ABE [8, 16, 38, 39, 59, 70] is the most expressive among them, though it comes with limitations with respect to fine-grained access and and dynamic updates [35]. Current ABE-based systems lack homomorphic capabilities (i.e., no computation on ciphertexts) and scalability required for time series data workloads. In general, adding homomorphic capabilities to ABE remains an open challenge [22]. Recently, important progress has been made on constructions of homomorphic attribute based encryption [20, 22, 30, 36]. However, they remain limited in functionality and are computationally expensive. A related line of work is searching over encrypted data with predicate evaluation [21, 76]. While predicate encryption schemes [21, 76] support range queries over encrypted data, they lack the required efficiency in our setting, as they require a linear scan through the database and also due to their underlying computationally expensive pairing-crypto.

8 Conclusion

In this paper, we presented TimeCrypt, a new scalable system that enables fast analytics over large encrypted data streams. TimeCrypt introduces HEAC, a novel encryption construction that enables execution of real-time analytics over encrypted stream data and empowers data owners to enforce access restrictions on encrypted data based on their privacy and access control preferences. Our evaluation on various large-scale workloads shows TimeCrypt's performance is close to operating on plaintext data, demonstrating the feasibility of providing high-performance and strong confidentiality guarantees when operating on large-scale sensitive time series data.

Acknowledgments

References

[1] Amazon Kinesis Access Control. Online: https://docs.aws.amazon.com/streams/latest/dev/controlling-access.html.

[2] Datadog DevOps. Online: https://www.datadoghq.com/solutions/devops/.

[3] Fraunhofer-AISEC ABE Library. Online: https://github.com/Fraunhofer-AISEC/rabe.

[4] Graphite. Online: https://graphiteapp.org.

[5] Thingsboard. Online: https://thingsboard.io/.

[6] Gergely Ács and Claude Castelluccia. I Have a DREAM! (DiffeRentially privatE smArt Metering). In *International Workshop on Information Hiding*, pages 118–132. Springer, 2011.

[7] Nitin Agrawal and Ashish Vulimiri. Low-Latency Analytics on Colossal Data Streams with SummaryStore. In *ACM SOSP*, 2017.

[8] Shashank Agrawal and Melissa Chase. FAME: Fast Attribute-based Message Encryption. In *ACM CCS*, 2017.

[9] Shweta Agrawal and Dan Boneh. Homomorphic MACs: MAC-Based Integrity for Network Coding. In *ACNS*, pages 292–305. Springer, 2009.

[10] Amazon. Amazon Timestream. Online: https://aws.amazon.com/timestream/.

[11] Amazon. Time Series Processing. Online: https://media.amazonwebservices.com/architecturecenter/AWS_ac_ra_timeseriesprocessing_16.pdf.

[12] Michael P. Andersen and David E. Culler. BTrDB: Optimizing Storage System Design for Timeseries Processing. In *USENIX FAST*, 2016.

[13] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for Key Management Part 1: General (revision 3). *NIST special publication*, 2012.

[14] Andrew Baumann, Marcus Peinado, and Galen Hunt.

Shielding Applications from an Untrusted Cloud with Haven. *ACM TOCS*, 33(3):8, 2015.

[15] Julius S Bendat and Allan G Piersol. *Random Data: Analysis and Measurement Procedures*, volume 729. John Wiley & Sons, 2011.

[16] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-Policy Attribute-Based Encryption. In *IEEE Security and Privacy*, 2007.

[17] John Biggs. It's time to build our own Equifax with blackjack and crypto. Online. http://tcrn.ch/2wNCgXu, September 2017.

[18] Biovotion. Medical Grade Health Monitoring Wearable. Online: http://www.biovotion.com/.

[19] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *ACM CCS*, 2017.

[20] Dan Boneh, Craig Gentry, Sergey Gorbunov, Shai Halevi, Valeria Nikolaenko, Gil Segev, Vinod Vaikuntanathan, and Dhinakaran Vinayagamurthy. Fully Key-Homomorphic Encryption, Arithmetic Circuit ABE, and Compact Garbled Circuits. Cryptology ePrint Archive, Report 2014/356, 2014.

[21] Dan Boneh and Brent Waters. Conjunctive, Subset, and Range Queries on Encrypted Data. In *TCC*, 2007.

[22] Zvika Brakerski, David Cash, Rotem Tsabary, and Hoeteck Wee. Targeted Homomorphic Attribute-Based Encryption. In *TCC*, 2016.

[23] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. [Extended Version] TimeCrypt: Encrypted Data Stream Processing at Scale with Cryptographic Access Control. *In ArXiv, arXiv:1811.03457v2 [cs.CR]*, 2020.

[24] Barbara Carminati, Elena Ferrari, and Kian Lee Tan. Specifying Access Control Policies on Data Streams. In *DASFAA*, pages 410–421. Springer, 2007.

[25] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: data structures and implementation. In *NDSS*, 2014.

[26] Cassandra database. Online: http://cassandra.apache.org/.

[27] Claude Castelluccia, Aldar CF Chan, Einar Mykletun, and Gene Tsudik. Efficient and Provably Secure Aggregation of Encrypted Data in Wireless Sensor Networks. *ACM TOSN*, 5(3):20, 2009.

[28] Dario Catalano and Dario Fiore. Practical Homomorphic MACs for Arithmetic Circuits. In *EUROCRYPT*, pages 336–352. Springer, 2013.

[29] Long Cheng, Fang Liu, and Danfeng Daphne Yao. Enterprise Data Breach: Causes, Challenges, Prevention, and future Directions. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(5), 2017.

[30] Michael Clear and Ciaran McGoldrick. Multi-identity and Multi-key Leveled FHE from Learning with Errors. In *CRYPTO*, 2015.

[31] Aloni Cohen, Shafi Goldwasser, and Vinod Vaikuntanathan. Aggregate Pseudorandom Functions and Connections to Learning. In *TCC*, 2015.

[32] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *USENIX NSDI*, 2017.

[33] Ketan Duvedi, Jinhua Li, Dhruv Garg, and Philip Ogden. Netflix: Scaling Time Series Data Storage. Medium, Online: https://medium.com/netflix-techblog/ec2b6d44ba39, January 2018.

[34] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K Cunningham. Sok: Cryptographically protected Database Search. In *IEEE Symposium on Security and Privacy*, 2017.

[35] W. C. Garrison, A. Shull, S. Myers, and A. J. Lee. On the Practicality of Cryptographically Enforcing Dynamic Access Control Policies in the Cloud. In *IEEE Security and Privacy*, 2016.

[36] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *CRYPTO*, 2013.

[37] Google. Protocol Buffers. Online: https://developers.google.com/protocol-buffers/.

[38] Vipul Goyal, Abhishek Jain, Omkant Pandey, and Amit Sahai. Bounded Ciphertext Policy Attribute Based Encryption. In *ICALP*, 2008.

[39] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based Encryption for Fine-grained Access Control of Encrypted Data. In *ACM CCS*, 2006.

[40] Trinabh Gupta, Rayman Preet Singh, Amar Phanishayee, Jaeyeon Jung, and Ratul Mahajan. Bolt: Data Management for Connected Homes. In *USENIX NSDI*, 2014.

[41] Isabelle Hang, Florian Kerschbaum, and Ernesto Damiani. ENKI: Access Control for Encrypted Query Processing. In *ACM SIGMOD*, 2015.

[42] InfluxDB. Online: https://www.influxdata.com/.

[43] InfluxDB Data Retention. Online: `https://docs.influxdata.com/influxdb/v1.7/guides/downsampling_and_retention/`.

[44] Java Netty Framework. Online: `https://netty.io/`.

[45] KairosDB. Online: `https://kairosdb.github.io/`.

[46] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-End Integrity Protection for Web Applications. In *IEEE Security and Privacy*, 2016.

[47] Alan F Karr, Xiaodong Lin, Ashish P Sanil, and Jerome P Reiter. Secure Regression on Distributed Databases. *Journal of Computational and Graphical Statistics*, 2005.

[48] Wilhelm Kleiminger, Christian Beckel, and Silvia Santini. Household occupancy monitoring using electricity meters. In *ACM UbiComp 2015*, Osaka, Japan, 2015.

[49] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler. JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT. In *USENIX Security*, 2019.

[50] Florian Lautenschlager, Michael Philippsen, Andreas Kumlehn, and Josef Adersberger. Chronix: Long Term Storage and Retrieval Technology for Anomaly Detection in Operational Data. In *USENIX FAST*, 2017.

[51] Ben Manes. Coffein: Java Caching Library. Online: `https://github.com/ben-manes/caffeine`, 2018.

[52] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient Private Statistics with Succinct Sketches. In *NDSS*, 2016.

[53] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. Grecs: Graph encryption for approximate shortest distance queries. In *ACM CCS*, 2015.

[54] Richard Mortier, Jianxin Zhao, Jon Crowcroft, Liang Wang, Qi Li, Hamed Haddadi, Yousef Amar, Andy Crabtree, James Colley, Tom Lodge, and et al. Personal Data Management with the Databox: What's Inside the Box? In *ACM CAN*, 2016.

[55] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *ACM CCS*, 2015.

[56] Lily Hay Newman. The Alleged Capital One Hacker Didn't Cover Her Tracks. WIRED, Online: `https://www.wired.com/story/capital-one-hack-credit-card-application-data/`, July 2019.

[57] OpenSSL Software Foundation. Online: `https://www.openssl.org/`.

[58] OpenTSDB. Online: `http://opentsdb.net/`.

[59] Rafail Ostrovsky, Amit Sahai, and Brent Waters. Attribute-based Encryption with Non-monotonic Access Structures. In *ACM CCS*, 2007.

[60] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big Data Analytics over Encrypted Datasets with Seabed. In *USENIX OSDI*, 2016.

[61] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind Seer: A Scalable Private DBMSs. In *IEEE Security and Privacy*, 2014.

[62] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *VLDB*, 8(12):1816–1827, 2015.

[63] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *ACM SOSP*, 2011.

[64] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A Secure Database using SGX. In *IEEE Security and Privacy*, 2018.

[65] Prometheus. Online: `https://prometheus.io/`.

[66] Jingjing Ren, Daniel J. Dubois, David Choffnes, Anna Maria Mandalari, Roman Kolcun, and Hamed Haddadi. Information Exposure From Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach. In *ACM IMC*, 2019.

[67] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security*, 2015.

[68] Alvin C. Rencher. *Linear Models in Statistics*. Wiley, 2007.

[69] Andrew Ross. The Connected Car Data Explosion: the challenges and opportunities. Information Age, Online: `http://www.bbc.com/news/technology-42853072`, January 2018.

[70] Amit Sahai and Brent Waters. Fuzzy Identity-Based Encryption. In *EUROCRYPT*, 2005.

[71] Sandeep Singh Sandha. StreetX: Spatio-Temporal Access Control Model for Data. *arXiv preprint arXiv:1711.03955*, 2017.

[72] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics

in the Cloud using SGX. In *IEEE Security and Privacy*, 2015.

[73] Mathew J. Schwartz. GDPR: Europe Counts 65,000 Data Breach Notifications So Far. Online: https://www.bankinfosecurity.com/gdpr-europe-counts-65000-data-breach-notifications-so-far-a-12489, May 2019.

[74] Hossein Shafagh, Anwar Hithnawi, Lukas Burkhalter, Pascal Fischli, and Simon Duquennoy. Secure Sharing of Partially Homomorphic Encrypted IoT Data. In *ACM SenSys*, 2017.

[75] Hossein Shafagh, Anwar Hithnawi, Andreas Dröscher, Simon Duquennoy, and Wen Hu. Talos: Encrypted Query Processing for the Internet of Things. In *ACM SenSys*, 2015.

[76] Elaine Shi, John Bethencourt, T.-H.H. Chan, Dawn Song, and Adrian Perrig. Multi-Dimensional Range Query over Encrypted Data. In *IEEE Security and Privacy*, 2007.

[77] Brian Thorne. Java Paillier Library (javalier). Online: https://github.com/n1analytics/javallier, 2018.

[78] Timescale. Online: https://www.timescale.com/.

[79] Timescale. Time Series Benchmark Suite. Online: https://github.com/timescale/tsbs.

[80] Timescale Data Retention. Online: https://docs.timescale.com/v1.2/using-timescaledb/data-retention.

[81] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing Analytical Queries Over Encrypted Data. In *VLDB*, pages 289–300, 2013.

[82] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta N Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. FarmBeats: An IoT Platform for Data-Driven Agriculture. In *USENIX NSDI*, 2017.

[83] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. Sieve: Cryptographically Enforced Access Control for User Data in Untrusted Clouds. In *USENIX NSDI*, 2016.

[84] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. IntegriDB: Verifiable SQL for Outsourced Databases. In *ACM CCS*, 2015.