Secure Sharing of Partially Homomorphic Encrypted IoT Data

Hossein Shafagh, Anwar Hithnawi, Lukas Burkhalter, Pascal Fischli, Simon Duquennoy

Secure Sharing of Partially Homomorphic Encrypted IoT Data

Hossein Shafagh
Department of Computer Science
ETH Zurich, Switzerland
shafagh@inf.ethz.ch

Anwar Hithnawi
Department of Computer Science
ETH Zurich, Switzerland
hithnawi@inf.ethz.ch

Lukas Burkhalter
Department of Computer Science
ETH Zurich, Switzerland
burkhalter@inf.ethz.ch

Pascal Fischli
Department of Computer Science
ETH Zurich, Switzerland
fischli@inf.ethz.ch

Simon Duquennoy
Inria Lille - Nord Europe, France and
RISE SICS, Sweden
simonduq@sics.se

ABSTRACT

IoT applications often utilize the cloud to store and provide ubiquitous access to collected data. This naturally facilitates data sharing with third-party services and other users, but bears privacy risks, due to data breaches or unauthorized trades with user data. To address these concerns, we present Pilatus, a data protection platform where the cloud stores only encrypted data, yet is still able to process certain queries (e.g., range, sum). More importantly, Pilatus features a novel encrypted data sharing scheme based on re-encryption, with revocation capabilities and in situ key-update. Our solution includes a suite of novel techniques that enable efficient partially homomorphic encryption, decryption, and sharing. We present performance optimizations that render these cryptographic tools practical for mobile platforms. We implement a prototype of Pilatus and evaluate it thoroughly. Our optimizations achieve a performance gain within one order of magnitude compared to state-of-the-art realizations; mobile devices can decrypt hundreds of data points in a few hundred milliseconds. Moreover, we discuss practical considerations through two example mobile applications (Fitbit and Ava) that run Pilatus on real-world data.

KEYWORDS

Secure Sharing, Encrypted Data Processing, Homomorphic Encryption

ACM Reference format:
Hossein Shafagh, Anwar Hithnawi, Lukas Burkhalter, Pascal Fischli, and Simon Duquennoy. 2017. Secure Sharing of Partially Homomorphic Encrypted IoT Data. In *Proceedings of SenSys '17, Delft, Netherlands, November 6–8, 2017,* 14 pages.
https://doi.org/10.1145/3131672.3131697

1 INTRODUCTION

The Internet of Things (IoT), through embedded devices and wearables, is enabling a whole new spectrum of applications. One of the prominent offerings in this domain is the emerging field of health and fitness tracking with over 150k [19] applications listed in the two major smartphone app stores. Examples of such applications include wristbands that can infer stress level from skin conductance, sport trackers that can log physical activities, and fertility apps.

The collected data typically consists of sensor readings (e.g., body temperature, conductance response), activity meta-data (e.g., duration, type), or health-related symptoms (e.g., migraine headaches, pain). For scalability, ubiquitous access, and sharing possibilities, the data is most often stored in the cloud. Transparent and secure data sharing (e.g., sharing with friends or domain experts) is considered a key requirement for the practicality and success of typical IoT systems [31, 52]. Moreover, securing the cloud storage is of utmost importance, as the data can be used to infer privacy-sensitive information, such as heart diseases, personal well-being, and fertility-related data [5, 15]. The privacy risks of today's data collection model are many, including systematic unauthorized disclosure of personalized data on clouds [7], for personal advertising [43], trading with insurances [17], and due to external [20] or internal data breaches (e.g., curious cloud employees [20]).

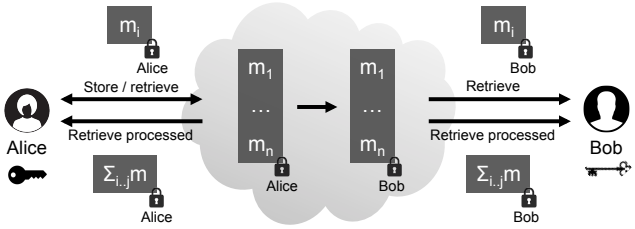


Figure 1: Pilatus can both query and share encrypted data. The cloud has access to no keys nor any plaintext. It is able to process encrypted data (e.g., range, sum queries), as well as to re-encrypt it, enabling crypto-protected sharing. We also address revocation, in situ re-keying, and group sharing.

Challenges. *How to benefit from cloud computing (i.e., storage and query processing) without compromising data control and security?* Storing encrypted data with traditional symmetric encryption schemes, such as AES, would offer protection but render the data unsearchable and unshareable. Alternatively, homomorphic encryption schemes enable arbitrary computation on encrypted data but are presently impractical [46]. In this paper, we focus on

SenSys '17, November 6–8, 2017, Delft, Netherlands

ACM 978-1-4503-5459-2/17/11…$15.00
https://doi.org/10.1145/3131672.3131697

Partially Homomorphic Encryption (PHE) and in particular additive homomorphic schemes. These are practical solutions that enable an important set of queries [62], such as the `sum` query on encrypted data – a common operation in IoT applications when history data needs summing or averaging. Also note that with limited involvement of the client side, more complex computations (e.g., linear regression) can also be achieved [62].

How to bring cryptographically guaranteed sharing to the IoT ecosystem? The current PHE approaches are either targeted at single-key encrypted data [46, 53, 62] (no support for sharing) or consider only text-based data [27, 47] (of limited use in an IoT context). Talos [51, 53] is specifically tailored for IoT scenarios and has demonstrated PHE on embedded devices, but it does not offer any sharing features. Existing protocols for sharing, such as OAuth [35], fall short in providing strong assurances about the policy enforcements. Crypto-based sharing approaches [65], on the other hand, support no query processing over encrypted data.

In short, existing solutions either support encrypted query processing or secure sharing but not both. Moreover, due to their heavy computational overhead, PHE schemes have been considered unsuitable for low-power mobile and IoT devices. In this paper, we tackle the challenges of cryptographically-protected and efficient sharing of PHE data, as illustrated in Figure 1. Our system is the first to combine encrypted sharing with encrypted query processing. Furthermore, our system is tailored towards mobile platforms, improving the state-of-the-art performance by more than one order of magnitude.

Approach. We introduce Pilatus, which extends Talos [53] with sharing capabilities. We enable efficient sharing of PHE data based on a re-encryption scheme [4]. This means that data is (PHE) encrypted at the client (IoT device/gateway) and uploaded to the cloud. When data owner Alice intends to share her data with Bob, she computes a token that enables the cloud to re-encrypt her data for Bob (without decrypting it first). The same process is used for Alice to share her data with a group. With only public keys and tokens (no secret keys nor plaintext information), the cloud is able to perform query and sharing operations directly on ciphertexts. Users can decide between sharing their individual data points (necessary for complex analytics) or aggregated results, both in encrypted form.

Further, we design a key revocation mechanism that allows users to terminate their data sharing at any time. We also propose an in situ key-update at the cloud, such that old data becomes protected with the owner's new key, without trusting the cloud with any private keys.

Our solutions build on the Elliptic Curve (EC)-based partially homomorphic encryption. Hence, a major challenge is to minimize the decryption time of our EC-based cryptosystems, for both sharing and encrypted processing. We address the performance optimization with the Chinese Remainder Theorem, which enables smaller, faster, and parallel computations. We keep the induced overheads low enough to preserve the user experience (i.e., below 1 s response time [41], including network latency) such that users can interact with their remotely stored data seamlessly.

Contributions. In summary, this paper makes the following contributions:

- We introduce a practical construction for sharing of PHE IoT data, with a sharing revocation mechanism that also allows in situ re-keying of the ciphertexts in the cloud;
- We improve the underlying cryptosystems' decryption time by one order of magnitude compared to state-of-the-art realizations such as Talos [53] and CryptDB [46];
- We design and implement Pilatus, a system that extends Talos with the above sharing schemes and with the necessary features for a practical sharing-enabled cloud storage;
- We assess the efficiency of Pilatus through end-to-end and micro benchmarks, in Amazon cloud, on mobile devices, and to a lesser extent on low-power sensor devices. We also present two case study apps running Pilatus: the Fitbit fitness tracker and the Ava fertility tracker [5]. We make our prototype implementation of Pilatus publicly available[1].

2 THREAT MODEL

We focus on IoT applications where data from wearables/smartphones are stored on third party clouds. We consider the following parties: the cloud (consisting of a front-end server and the database), clients (apps), and an Identity Provider (IDP) to certify the public key of each user.

Threats. Pilatus considers the cloud service to be honest-but-curious, such that it will follow the protocol correctly, but tries to learn as much as possible from the stored data. This is a valid model, as protocol violations, once detected, could penalize the service provider. At the same time, the adversary might be eager to learn more about user data without being noticed (i.e., passive). External adversaries can also gain access to the encrypted data as a consequence of system compromise. In summary, we consider cloud-side threats which are due to system compromise (e.g., data theft), financial incentives (e.g., unauthorized trades with users data), or malicious insiders (e.g., curious admins).

Assumptions. In addition to the honest-but-curious cloud assumption, Pilatus assumes that the IDP correctly verifies users' identity-key pairs. The IDP is a standard requirement in multi-user systems and can be a known and trustworthy external entity or an internal unit. Group members of shared data are semi-trusted, in that they do not collude with the cloud to leak the group data or key. This is a reasonable assumption for small groups where members are acquainted with each other. Moreover, Pilatus assumes that the applications behave correctly and do not hand out user keys to malicious parties. Finally, we assume state-of-the-art security mechanisms to be in place for device security and that all parties communicate over secure channels.

Guarantees. Pilatus protects the confidentiality of users data stored in the cloud in the presence of passive adversaries (e.g., compromised servers). In case the user device is compromised and group keys are disclosed, only data of the compromised user and the affected group are disclosed. Pilatus cryptographically prevents unauthorized data access. Pilatus provides user authentication mechanisms but does not guarantee freshness or correctness of

[1] Pilatus is available at https://github.com/Talos-crypto/Pilatus

the retrieved data. In order to provide such guarantees, one could complement our system with integrity protection frameworks such as Verena [32]. Pilatus does not hide access patterns, which can potentially reveal sensitive information [29].

3 PILATUS OVERVIEW

We briefly introduce the requirements of the applications we target and then present the architecture of Pilatus.

3.1 Applications

Pilatus focuses on applications collecting sensitive data that require processing and sharing. Examples of such applications include fitness or health trackers. These applications store private data in the cloud that can reveal privacy-sensitive information, e.g., illness, lifestyle, or location. For instance, Fitbit wristbands collect a user's heart rate, step counts, and location data. Similarly, certain health tracking applications and wearables, such as Ava [5], allow women to track their menstrual cycle, predict (in)fertile phases, and detect potential health issues. Though insightful, logging such private information raises serious privacy concerns.

Secure and transparent sharing plays an essential role in bringing such fitness- and health-related apps to their full potential. When users are willing to share data with experts (e.g., medical practitioners), analytical services, or just casually with friends, they must be in full control over who can access and what can be accessed. Moreover, there is a need for query processing capabilities directly in the cloud, since downloading the entire data volume for on-device processing becomes impractical as the data grows. For instance, in Fitbit and Ava, the mobile apps typically display total sums or averaged values (e.g., heart rate) over a given period of time.

To satisfy the above requirements, Pilatus facilitates storing encrypted IoT data in the cloud, while enabling processing and cryptographically-protected sharing of encrypted data. In §7.2, we show that our Pilatus-based app prototypes Ava and Fitbit induce only a modest overhead (i.e., a maximum of 130 ms) while interacting with encrypted data. Note that while the focus of this paper is on health and fitness applications, our system design and application discussions apply as well to other IoT applications (smart homes, connected cars, etc.).

3.2 Architecture

Pilatus, as an extension of Talos, uses Partially Homomorphic Encryption (PHE) schemes to enable query processing over encrypted data in the cloud. It offers a new scheme for secure sharing of PHE data among users and groups, based on re-encryption techniques. Our sharing feature includes access revocation with in situ re-keying. We optimize the performance of the underlying cryptographic schemes by one order of magnitude to make them practical on mobile devices. Note that Pilatus inherits the order-preserving encryption from Talos for range queries. Together with PHE, this enables querying a sum/average over a range of timestamps or any other relevant combination of metadata[2].

Pilatus consists of three main components: the client engine, the cloud engine, and an identity provider, as depicted in Figure 2.

[2]We apply range queries to data types that are of high entropy from a sparse domain to avoid any data leakage due to inference attacks [40].

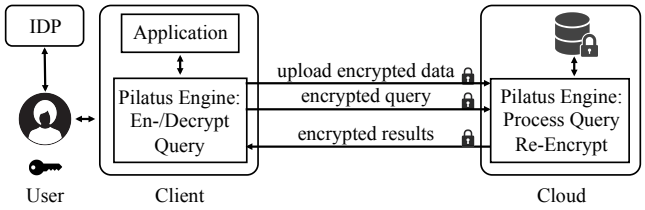


Figure 2: Pilatus architecture. The client engine performs en-/decryptions, such that the cloud only stores encrypted data. The cloud engine performs encrypted query processing and necessary re-encryptions for data sharing.

Client Engine. The client engine runs on the user side and is the only component with access to keying material. It is primarily designed for users' personal mobile platforms, such as smartphones. It interacts with the IDP to verify the ID-to-key bindings and with the cloud engine for secure storage, sharing, and retrieving of data. More specifically, it encrypts, decrypts, handles the keys, and sharing-related activities such as joining new groups, issuing delegated access rights (i.e., re-encryption tokens), triggering revocation, and re-keying. For constrained IoT devices (sensors), we have a stripped-down client engine with limited functionality, such as encryption to store PHE encrypted data in the cloud.

Cloud Engine. The cloud engine is application-agnostic and provides the basic database interface and features. It stores data and processes queries from the client engine. The cloud engine has only access to data in encrypted form. It supports the algorithms required for processing encrypted data, i.e., homomorphic addition, re-encryption, and in-situ re-keying. Our design is currently targeted for structural databases (i.e., MySQL) and uses User-Defined Functions to replace the default routines with crypto-enabled ones (see §6 for details).

Identity Provider. The IDP is an independent party responsible for verifying the user identity to public key bindings. The IDP is used by the client engine to search for the public key of another user or group. Pilatus is independent of the IDP and outsources this role to systems such as Keybase [34], that provide provable identity-key bindings, by utilizing prevalent social media channels and online accounts (i.e., users prove their identity by posting an individual token on Twitter, Facebook, Github, etc.).

4 CRYPTOGRAPHIC BACKGROUND

We present here the necessary cryptographic background to help understand the Pilatus design.

4.1 Partial Homomorphic Encryption

Partial Homomorphic Encryption (PHE) schemes allow the computation of certain mathematical operations over encrypted data. For instance, additive homomorphic schemes, such as the Paillier cryptosystem [44], support the addition of ciphertexts, such that the result is equal to the addition of the plaintext values (i.e., $ENC(m_1)\ o\ ENC(m_2) = ENC(m_1 + m_2)$).

Figure 3: Plaintext to EC point mapping before encryption and after decryption.

The Elliptic Curve (EC) version of the ElGamal cryptosystem is an alternative additive homomorphic scheme, used in Talos and Pilatus. EC-ElGamal's security is based on the EC Discrete Logarithm Problem (ECDLP) [57]. It provides semantic security (i.e., IND-CPA under the assumption of decisional Diffie-Hellman). A challenge in making practical use of EC-ElGamal is that it operates over EC points rather than arbitrary messages. Hence, one needs a scheme that maps an integer to an EC point (and back), while preserving the homomorphic property of EC-ElGamal.

Talos, which focuses on IoT data, uses a theoretical method [57] that becomes practical for small integer data, e.g., 32-bit (frequent in IoT scenarios, to represent an integer or fixed point number) [53]. The process is as follows: to map an integer m to an EC point M, m is multiplied by a publicly known point G on the curve: $M = mG$. After decryption, M must, however, be mapped back to m. This requires solving an ECDLP. Although this is computationally infeasible for large numbers, solving it for smaller than 32-bit integers can be realized in a reasonable time with, e.g., the Baby-Step-Giant-Step (BSGS) algorithm (this is equivalent to breaking 32-bit security). Note that this mapping procedure, as depicted in Figure 3, does not affect the overall security: the ECDLP is solved to obtain m from M, but M itself is protected with strong cryptography, in this case, 80-bit or 128-bit security.

Pilatus inherits this scheme from Talos, but presents additional optimizations to speed up the process by one order of magnitude, as discussed in §5.2.

4.2 Re-Encryption

Re-Encryption (RE) enables a proxy to convert ciphertexts under Alice's key to ciphertexts under Bob's key, without disclosing the plaintext. Hence Alice can share data with Bob, without sharing her private key nor performing any encryption for Bob on her personal device.

The AFGH [4] RE scheme relies on pairing-based cryptography. AFGH defines next to the standard functions *key generation*, *encryption*, and *decryption*, two additional functions: *re-encryption-token generation* and *re-encryption*. The former is used by Alice to generate the re-encryption token for Bob, based on her own private key and Bob's public key. The latter performs the re-encryption from Alice to Bob given the ciphertext is encrypted under Alice's public key.

At a higher abstraction, pairings (or bilinear maps) establish a relationship between two cryptographic groups. In AFGH, re-encryption consists in transforming a ciphertext from the first group to the second group. The underlying pairing in AFGH is essentially a bilinear map [11] which, given a cyclic group $\mathbb{G}$ of prime order q, has the following property for $a, b \in \mathbb{Z}q$ and $g, h \in \mathbb{G}$: $e(g^a, h^b) = e(g, h)^{ab}$. Such maps can be realized with the Weil and Tate pairings, which are efficiently computable with Miller's algorithm [39]. However, designing efficient pairings is an ongoing research topic [2].

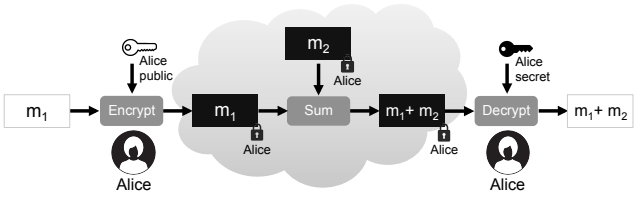


Figure 4: Encrypted query processing. The data is encrypted with Alice's public key. Computations take place in the cloud, on ciphertexts. The result is decrypted with Alice's private key.

More formally, AFGH defines the system parameters as (g, e, Z, $\mathbb{G}$, $\mathbb{G}_t$), where $g \in \mathbb{G}$, $Z = e(g, g) \in \mathbb{G}_t$ and e as the map: $\mathbb{G}\times\mathbb{G} \rightarrow \mathbb{G}_t$. $\mathbb{G}$ and $\mathbb{G}_t$ are both cyclic groups of the same prime order. The user Alice computes her public key as $pk_a = g^a$ and her private key as $sk_a = a$. Alice can issue Bob the re-encryption token based on his public key pk_b as the $Token_{a\rightarrow b} = pk_b^{1/a} = g^{b/a} \in \mathbb{G}$. The encryption of m is performed as:

$$C_a = (mZ^r, g^{ar}) \tag{1}$$

where r is a random number. Note that the ciphertext C_a is composed of two components, similar to the EC-ElGamal ciphertext. A proxy with access to $Token_{a\rightarrow b}$ performs the re-encryption by transforming the second component of C_a:

$$C_b = (mZ^r, Z^{br}), \text{ with } Z^{br} = e(g^{ar}, g^{b/a}) \tag{2}$$

Bob can now decrypt the ciphertext C_b with his private secret $sk_b = b$ and the pairing Z as:

$$m = \frac{mZ^r}{(Z^{br})^{1/b}} \tag{3}$$

In §5.1.2, we show how to employ AFGH in an efficient additive homomorphic context. The cloud serves as the proxy, in charge of re-encrypting data of a user to another user or group.

5 PILATUS DESIGN

This section presents our EC-based encryption for sharing of PHE data and discusses aspects such as performance optimization, revocation, and authorization.

5.1 Processing and Sharing Encrypted Data

We present Pilatus's two modes of operation: Standard Mode and Data Sharing. The former covers the single-key case, while the latter enables cryptographically-protected sharing. Both modes exhibit additive homomorphic properties, as illustrated in Figure 4.

5.1.1 Standard Mode. When uploading data that is not intended for sharing, the Pilatus client engine selects the standard mode. The standard mode is mostly inherited from Talos and uses EC-Elgamal as an additive homomorphic encryption scheme (§4.1). However, the decryption in Talos suffers from an exponential increase of computational costs for larger integer values, as shown in Figure 6. In §5.2, we introduce our optimizations to overcome this shortcoming and accelerate decryption and enable the use of integers larger than 32-bit as plaintext.

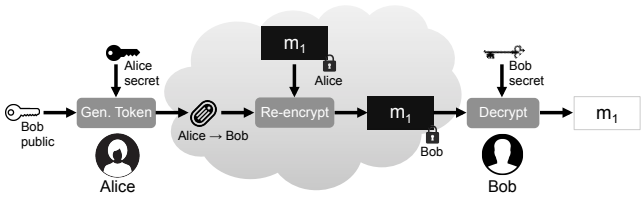


Figure 5: Data sharing (re-encryption). Alice generates a token, from her private key and Bob's public key. The cloud uses the token to re-encrypt Alice's data as Bob's data. Bob can in turn decrypt with his private key. The same mechanism is used for group sharing. Note that re-encryption is non-transitive, i.e., the result can not be re-encrypted again.

5.1.2 Data Sharing. In Pilatus, we construct an efficient additive homomorphic interpretation of AFGH (presented in §4.2). This enables cryptographically-protected (as opposed to policy-based) sharing of PHE data, without the need to disclose any private keys to the cloud. All the cloud needs is a token generated by the owner, from the owner's private key and the target user/group's public key, as illustrated in Figure 5.

To realize the homomorphic additive property, we use the algebraic structure of elliptic curves over finite fields, similar to [64]. Note that bilinear-map-based cryptosystems, such as AFGH, leave the selection of the underlying pairing-friendly elliptic curve to implementation. In the recent years, research on optimal paring curves [21] has further progressed. In Pilatus, we rely on the optimal Ate pairing [63][3].

To enable the additive homomorphic property, we represent message m as $M = Z^m$, with Z as the pairing. Given the public key $pk_a = g^a$ of Alice with g as a generator point in $\mathbb{G}$ and the random r, we encrypt as:

$$C_a = (Z^m Z^r, pk_a^r) = (Z^{m+r}, g^{ar}) \tag{4}$$

and re-encrypt for Bob as:

$$C_b = (Z^{m+r}, Z^{br}), \text{ with } Z^{br} = e(g^{ar}, g^{b/a}) \tag{5}$$

With access to the private key b, Bob can decrypt as:

$$M = \frac{Z^{m+r}}{(Z^{br})^{1/b}} = \frac{Z^{m+r}}{Z^r} = Z^m \tag{6}$$

Note that in the final step of decryption, we still need to map back M to m (i.e., solving a discrete log problem), which similar to the standard mode benefits from our performance optimization, presented in the next section.

The homomorphic addition of two ciphertexts C_{b1} and C_{b2} (encrypted under Bob's key) is performed as follows:

$$\begin{aligned} C_{b1} + C_{b2} &= (Z^{m1+r1}, Z^{br1}) \odot (Z^{m2+r2}, Z^{br2}) \\ &= (Z^{m1+r1} Z^{m2+r2}, Z^{br1} Z^{br2}) = (Z^{m1+r1+m2+r2}, Z^{br1+br2}) \\ &= (Z^{m1+m2+r1+r2}, Z^{b(r1+r2)}) \end{aligned} \tag{7}$$

Because AFGH is key-optimal, Pilatus's storage size for a user remains constant regardless of the number of users/groups it shares the data to. Moreover, the re-encryption tokens are unidirectional and non-transitive. This implies, given $Token_{a\to b}$ it is only possible to re-encrypt Alice's ciphertexts C_a to Bob's ciphertexts C_b. The opposite direction is cryptographically not feasible. Additionally, given $Token_{a\to b}$ and $Token_{b\to m}$ it is cryptographically not feasible to transform Alice's ciphertext to Mallory's ciphertext.

[3]The optimal Ate pairing is over Barreto-Naehrigopera curves [2, 8], which are pairing-friendly elliptic curves of prime order, with embedding degree 12.

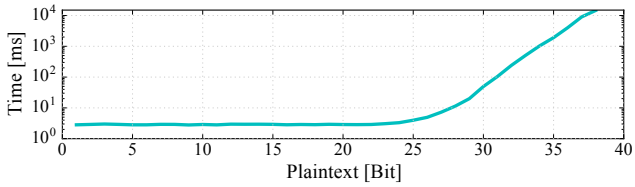


Figure 6: Decryption in EC-ElGamal with the Baby-Step-Giant-Step algorithm, 80-bit security and 4 threads on Nexus 5. Smaller plaintexts (23-bit) are decrypted efficiently (<3 ms), larger plaintexts cause an exponential slowdown.

5.2 Performance Optimization

At decryption, the EC mapping proposed by Talos (see §4.1) requires solving an ECDLP problem to convert a plaintext EC point to the original plaintext integer. As discussed earlier, using the Baby-Step-Giant-Step (BSGS) algorithm we can solve the ECDLP for small integer values (i.e., ≤32-bit) within a few milliseconds. However, the performance of this algorithm decreases exponentially with larger integers, as depicted in Figure 6 (e.g., already 15 s for decryption of 38-bit integers). This is because with each additional bit the search space is doubled until it becomes too large to efficiently find the solution (i.e., computing the discrete logarithm).

This technique has two major shortcomings with regards to our design: *(i)* batch decryption can harm user experience, exceeding 1 s with as few as 25 decryptions (32-bit values); *(ii)* with larger numbers, e.g., 64-bit integers, this approach becomes impractical. This demands an optimization that also maintains the homomorphic property of these schemes.

Approach. Our optimization is based on combining the Chinese Remainder Theorem (CRT) [28] with the BSGS algorithm. It is applicable for decryption in both the standard and sharing modes. With CRT, we reduce solving one difficult ECDLP problem (i.e., a large integer) to solving several smaller ECDLP problems, for which the BSGS algorithm performs efficiently, as illustrated in Figure 7. As BSGS exhibits an exponential cost, our approach can provide drastic performance improvements.

CRT is used in many cryptographic constructions [16, 57] and Hu et al. [28] present a formal treatment on how to leverage CRT for homomorphic encryption schemes. We are the first to utilize the general CRT optimization in *combination* with the BSGS algorithm for an efficient computation of discrete logarithms in a pairing-based re-encryption.

The CRT technique is based on the simple idea of representing a number X uniquely by its remainders a_i from the following congruence equations, where n_i are co-primes (i.e., $\gcd(n_i, n_j) = 1$, $\forall\, i, j$):

$$X \cong a_i \bmod n_i \tag{8}$$

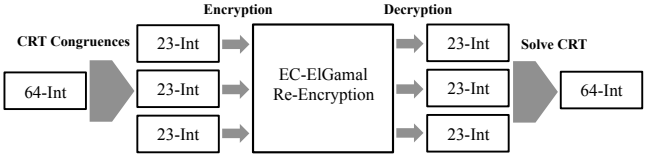


Figure 7: We optimize the decryption in both standard and sharing modes, with a technique based on the Chinese Remainder Theorem (CRT) where a larger value is represented by several smaller ones.

Hereby, N which is the product of all co-primes n_i (i.e., $N = \prod n_i$) should be larger than X. With regards to our encryption schemes (i.e., EC-ElGamal and pairing-based re-encryption), X corresponds to the plaintext value which can now be represented with the remainders a_i. Since the remainders are significantly smaller than X, the decryption is performed more efficiently. Given the co-primes n_i and the remainders a_i, X can be efficiently computed, as:

$$X = \sum_{i=1}^{r} a_i N_i y_i (mod\ N) \tag{9}$$

where $N_i = N/n_i$ and $y_i = N_i^{-1} mod\ n_i$. Note that y_i and N_i remain unchanged for a given set of co-prime values n_i and hence can be pre-computed in advance.

Realization. To utilize CRT to our benefit, we first compute the remainders a_1 and a_2 (assuming two congruences) for a given value X (i.e., as defined in Equation 8). Now instead of encrypting X, we encrypt a_1 and a_2. Note that the co-prime values n_1 and n_2 are public information.

Utilizing CRT has the side effect of increased encryption cost and ciphertext-size (linearly by the number of congruences). For instance, with 160-bit ECC (80-bit security), the ciphertext-size of EC-ElGamal increases for 32-bit values from 42 bytes (2 compressed EC-points) to 84 bytes (4 compressed EC-points), which is still lower than the ciphertext-size in the Paillier cryptosystem (256 bytes with 80-bit security).

The advantage of CRT becomes apparent while performing decryptions. Instead of solving ECDLP for a large X (which can take seconds to hours for larger values), we now solve it efficiently for the remainders (i.e., a_1, a_2). The larger the plaintext value, the higher the performance gain due to CRT (i.e., several orders of magnitude for large values). For instance, the 15 s decryption time for a 38-bit value is reduced to less than 10 ms for EC-ElGamal (see §7.1.2).

Homomorphism. We discuss here how we keep the additive homomorphic property with our CRT extension. To add two large integers X_a and X_b, we add their remainders (a_i and b_i respectively) in the corresponding congruences, as follows:

$$X_a + X_b = \sum_{i=1}^{r} (a_i + b_i) N_i y_i \ (mod\ N) \tag{10}$$

This is possible due to modular arithmetic ($X_a + X_b = a_i + b_i\ mod\ n_i$). Since our underlying encryption schemes are additive homomorphic, we can compute the addition of the corresponding encrypted remainders, as follows:

$$\begin{aligned} ENC(X_1) + ENC(X_2) = \\ (ENC(a_1), ENC(a_2)) + (ENC(b_1), ENC(b_2)) = \\ ENC(a_1 + b_1), ENC(a_2 + b_2) \end{aligned} \tag{11}$$

Hence, EC-ElGamal and our pairing-based re-encryption remain additive homomorphic.

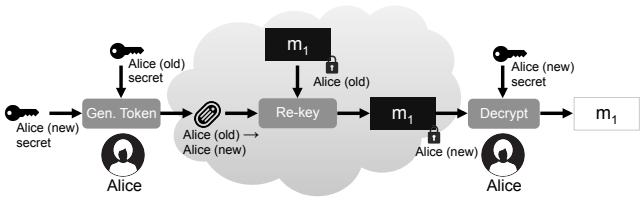


Figure 8: Data revocation (in situ re-keying). Alice builds a token from both her old and new private keys (none of which is leaked). The cloud uses the token to re-key Alice's data, such that they can now be decrypted with the new key. Alice can re-key multiple times and apply re-encryption to re-keyed data.

5.3 Key Revocation

To authorize data sharing, a data owner issues a cryptographic token, used by the cloud to re-encrypt users data towards the destination user. We address here the challenge of terminating such a data sharing, cryptographically.

Key Update. In Pilatus, when users decide to revoke a data sharing, they simply begin using new keys for new data. This renders previously issued tokens obsolete and prevents new ciphertexts cryptographically from being re-encrypted with the old token. Once new keys are in place, valid sharing relationships are updated with new tokens such that the sharing flow can be maintained. We discuss in §5.4 in more details how the data sharing authorization works with regards to joining and leaving groups. Note that a key revocation event can as well occur, when the encryption key of the user is compromised.

We consider two cases for the key update: malicious cloud and semi-honest cloud. In the former case, we leave old data protected with old keys. We consider such old data to be in the wild, since already shared and possibly cached at sharing parties. However, in the latter case, it is desirable to update the encryption keys of the old data to the latest keys, for consistent access. Hence, it is important to devise a secure solution that allows the semi-honest cloud to perform the re-keying without access to any private key.

In Situ Re-keying. To construct our re-keying mechanism, we leverage the fact that user Alice has access to both old and new private keys (a and a', respectively), neither of which is disclosed to any party[4]. Hence, at the time of re-keying, the private keys of Alice are not compromised.

[4]Note that in contrast to key homomorphic PRFs (i.e., Pseudo Random Functions) [65], where a symmetric key is shared between parties, our re-keying scheme is key-private.

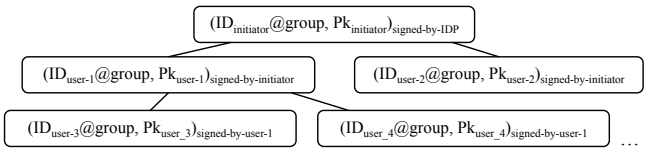


Figure 9: Access graph for group membership.

The re-keying (see Figure 8) is carried out on data, at the cloud, prior to sharing (i.e., ciphertexts in level-1, as represented in Equation 4). To this end, Alice issues a key-update token $\delta = a'/a$ and the cloud performs the re-keying as follows (only the second component of the ciphertext is adjusted):

$$C_{a'} = (C_1, \delta C_2) = (C_1, ra'G) \tag{12}$$

After re-keying, all ciphertexts on the cloud are encrypted with the latest key a'. Note that re-keying, unlike re-encryption, is transitive, i.e., the re-keying can be applied multiple times to the same item. The scheme would, however, be poorly suited for sharing, as it requires both the source and target private keys.

Note that since both old and new keys are only known to Alice, a curious cloud cannot learn anything about the private keys from δ. However, a malicious cloud could use the reverse of the key-update token (i.e., a/a') and downgrade ciphertexts encrypted under newest keys to old keys. This is why we only enable re-keying in case a semi-honest cloud or a trusted proxy is present.

5.4 Group Sharing Authorization

Data in Pilatus can be shared either directly with a user or a group. We describe a simple sharing authorization mechanism for group-related operations. The construction of the authorization is crucial as it ensures that a joining group member *(i)* issues the re-encryption token for the authentic group; and *(ii)* retrieves the correct group key. These two aspects are related, since the correct group key is necessary for the process of generating the re-encryption token.

Access Graphs. In our construction, we leverage access graphs, which are similar to certification paths in the public key infrastructure, in combination with our re-encryption scheme. We form an access graph for each group. The root node holds the initiator's ID concatenated with the group identity (e.g., "runners") and the public key of the initiator, as depicted in Figure 9. The joining members of the group compose the access graph nodes. Each node, except for the root, is signed by the immediate parent node. The root is signed by the IDP. The access graph allows group members to vouch for the membership of other members.

Joining. Group membership is authorized by a group member in two steps: *(i)* signing the extended identity (i.e., ID@group name) and the public key of the new member, e.g., Alice, (after verifying the key correctness over the IDP); and *(ii)* issuing a re-encryption $Token_{g \to a}$ which is then stored in the cloud. After joining the group, Alice provides the re-encryption $Token_{a \to g}$, required for sharing with the group. To issue this re-encryption token and later be able to access (i.e., decrypt) group data, Alice needs the public (PK_g) and private (SK_g) keys of the target group. This information is stored signed and partly encrypted on the cloud:

$$(ID_g, PK_g, ENC_g(SK_g))_{signed-by-initiator}$$

Note that the initiator's signature on the key information prevents a malicious entity in deceiving Alice into issuing a token for a fake group with the same name. Currently, group members are authorized to add new members. We can restrict this authority to the initiator only by encrypting the group's private key with the initiator's key. A new member would require the initiator's authorization which is expressed in the $Token_{initiator \to a}$ to access the group key.

Leaving. To leave a group, Alice initiates our revocation procedure (c.f. §5.3). After revocation, new coming data is encrypted with new keys and can no longer be transformed with the expired $Token_{a \to g}$. Her previous data, however, remains in the wild and can still be accessed by the group members, unless Alice decides to trigger our in-situ re-keying feature. Note that disclosure of the group's secret key SK_g and Alice's $Token_{a \to g}$ does not expose Alice's private key.

Cryptosystem	Setup time [ms]	
	80-bit sec.	128-bit sec.
Paillier	1623	42190
Standard	0.28	0.61
Sharing: Key setup	4.15	6.72
Sharing: Token gen.	2.5	4.7

Table 1: Average setup time on Nexus 5 for 80-bit and 128-bit security levels.

5.5 Security Analysis

Our main security goals are to defy passive attacks targeted at data on the cloud as well as to prevent access of unauthorized users (see §2). Pilatus achieves these goals such that data on the cloud remains strongly encrypted (i.e., semantic security) at all times. The cloud never gains access to any decryption keys. We rely on the IDP to prevent fake user creations.

To protect the data from unauthorized access, we cryptographically restrict data access to users with decryption keys (i.e., either individual or group keys). With the re-encryption token, the cloud can only re-encrypt the stored data towards the authorized group/service. Moreover, we prevent a malicious cloud from performing unauthorized re-encryptions towards a malicious user (thanks to the one-hop property of the re-encryption scheme).

The disclosure of group keys does not affect the security of the corresponding private keys of the group members. This is because our underlying re-encryption scheme is key-private. After such an incident, members can perform a revocation to terminate the data transformation into group data.

Our in situ key-update technique assumes a semi-honest cloud. In other cases, re-keying can either be disabled or delegated to a trusted proxy.

Mode	Security	Smartphone			Constrained IoT	Cloud		
		ENC [ms]	DEC-1 [ms]	DEC-2 [ms]	ENC [ms]	ADD-1 [μs]	ADD-2 [μs]	Share [ms]
Standard	80	2.4	2	-	252	54	-	-
	128	4.7	3.9	-	530	91.8	-	-
Sharing	80	9.8	11.9	9	not available	62	30.6	1.7
	128	15.2	17.4	13.3	not available	75	73.7	2.3

Table 2: Complete overview of our evaluation results for 32-bit integers (i.e., 2 CRTs) with 80-bit security. DEC-1/2 and ADD-1/2 refer to operations (1) before and (2) after sharing. Note that on smartphones, batch encryption/decryption with multithreading (i.e., 4 threads) yields us a 3× performance improvement.

6 IMPLEMENTATION

We implemented a prototype of Pilatus for mobile platforms (user/client devices) and the cloud (structured data storage). The client engine (Android) consists of a REST client for interactions with the cloud. Application developers can use the Pilatus API which internally calls their previously defined SQL procedures. The client engine handles data encryption before performing requests and decryption after retrieving the data from an API call.

For the EC-ElGamal encryption, we utilize the ECC module of the OpenSSL library (v1.1.0). We implemented the data sharing components (i.e., re-encryption) based on the RELIC toolkit [1, 2]. We support 80-bit and 128-bit security levels. Our BSGS algorithm implementation relies on hash-map (i.e., klib library) for the look-up table.

The cloud engine supports a MySQL database, for which we implemented the corresponding User Defined Functions (UDF). We use UDFs to replace the default routines with crypto-enabled ones, without the need of recompiling the database. Incoming queries indicate the UDF to be used, e.g., `SELECT SUM_EC_ELGAMAL(column-x) FROM table-y`, where the standard `SUM` is replaced with the dedicated homomorphic addition sum for EC-ElGamal. Moreover, the cloud engine is equipped with a REST engine (i.e., Restlet library).

The implementation of Pilatus consists of 2000 sloc of C/C++, 10000 sloc of Java, and another 4000 sloc for testing, setup scripts, and benchmarking. Our prototype Android applications Fitbit and Ava consist of 2400 and 2500 sloc, respectively.

Example Applications. To show the feasibility of Pilatus and evaluate its end-to-end performance, we developed two example mobile applications that integrate Pilatus, where the cloud components are hosted on Amazon's cloud services. Our Android activity tracking app operates on data collected by our personal Fitbit device. It fetches the data from Fitbit servers and stores it encrypted in our cloud instance. For our Ava fertility tracking app, we received anonymized data from the Avawomen startup [5]. In both apps, the users can interact with the data similar to the original apps. Our applications do not cache any data locally which allows us to study the worst-case performance while interacting with remote data. The cloud and the client communicate over HTTPS and data is encoded in JSON format, as a compact data representation form. For authentication, we rely on the OpenID Connect [35], where we currently support Google accounts [25] as a proof-of-concept.

Constrained IoT Devices. We implemented a prototype of Pilatus for more constrained IoT devices, where the client-engine only accommodates the encryption logic in the standard mode. We based our implementation on Contiki [18], an open-source low-power operating system for IoT devices. For cryptographic processing, we utilize both software libraries (i.e., RELIC toolkit [1]) and the hardware crypto accelerator.

7 EVALUATION

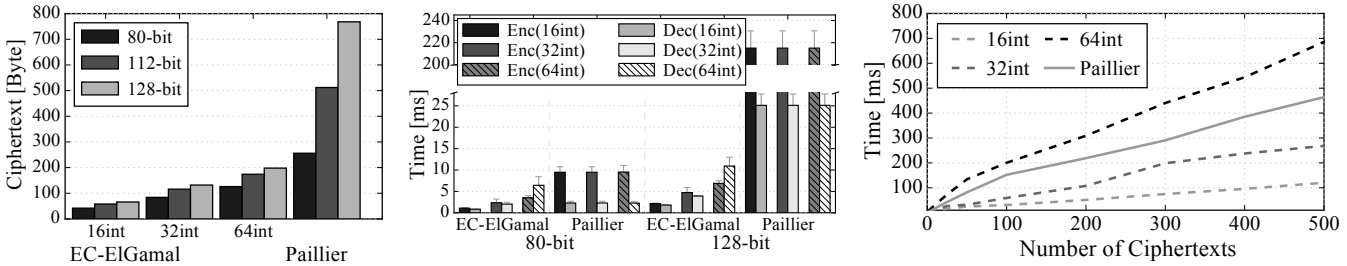
This section presents a thorough evaluation of Pilatus, both on the cloud and on client sides (mobile device and, to a lesser extent, constrained IoT device).

Evaluation Setup. Our evaluation setup consists of the client engine running on a smartphone and the cloud engine running at Amazon cloud services (AWS). We use an LG Nexus 5, equipped with a 2.3 GHz quad-core 64-bit processor and 2 GB RAM, running Android 5.1.1. Our AWS account provides 25 GB of storage and one instance of Intel Xeon 3.3 GHz CPUs with 1 GB RAM. For micro-benchmarks of homomorphic addition, we additionally use a MacBook Pro equipped with 2.2 GHz Intel Core i7 and 8 GB of RAM.

For the client engine, we also present results on constrained IoT devices with our Contiki implementation. We select OpenMotes as the hardware platform, which utilize the same class of MCU as activity trackers such as Fitbit. OpenMotes are based on the TI CC2538 microcontroller [60], i.e., 32-bit ARM Cortex-M3 at 32 MHz, with a public-key crypto accelerator running up to 250 MHz. They are equipped with 802.15.4-compliant RF transceivers, 32 kB of RAM and 512 kB of ROM.

Metrics. We rely on the following metrics to report on the overheads and performance of Pilatus:

- **Computation time** indicates the CPU time required to perform a certain operation. The computation time has a direct impact on the application delay and energy consumption of mobile platforms.
- **Ciphertext size** is an important metric considering network bandwidth. It measures the impact of Pilatus on the communication requirements.
- **System throughput** is the rate of operations on encrypted data performed in the cloud.



(a) Ciphertext size (bandwidth/memory).

(b) Paillier vs. our optimized EC-ElGamal with 80-bit and 128-bit security levels.

(c) Batch decryptions of varying size with 80-bit security and 4x threads on Nexus 5.

Figure 10: On-device Standard Mode evaluation. Compares Paillier (utilized in CryptDB) with our optimized EC-ElGamal. With the CRT technique, our standard mode can differentiate between different input lengths.

- **Application latency** reflects the time from a client initiating a query to the client receiving and decrypting the results. It accounts for query processing at the client side, plus network latency and cloud processing time.

In the next sections, we continue with discussing the results of the system components benchmark and then elaborate on the end-to-end evaluation results. Table 2 summarizes the evaluation results.

7.1 Micro-Benchmark

In this section, we discuss the performance of our crypto primitives.

Methodology. We instrument our client engine app to run the operations in isolation. We generate plaintext inputs uniformly at random for three input sizes: 16, 32, and 64-bit integers. We repeat each experiment at least 1000 times, except for expensive key generation operations which are repeated 100 times. For the time measurements, we rely on the Google Guava *Stopwatch* class[5] with nanoseconds accuracy. We only enable multi-threading support of Android for batch decryptions. All other reported numbers are measured while running on a single thread.

We first assess CRT in detail and then proceed to evaluating the standard and data sharing modes on mobile devices.

7.1.1 CRT Optimization. With our CRT optimization (see §5.2), we represent a larger value through its smaller remainders (i.e., congruences). We discuss here the tradeoff between decryption performance and ciphertext size due to CRT.

Typical integer values are either 16, 32, or 64-bit. Note that for selecting the size of congruences, the sum of congruence sizes must be larger than the integer value to be represented. 16-bit values can be calculated efficiently without CRT (Figure 6). For 32-bit values, we can either select two 17-bit or three 11-bit congruences. For 64-bit values, we can also use three 22-bit or four 17-bit congruences.

Table 3 shows the decryption time as a function of the number of congruences, compared to the state-of-the-art Talos, which does not use the CRT optimization. In the 32-bit case, we can improve the 42 ms from Talos to 2 ms. In the 64-bit case, decryption is plain infeasible in Talos, but becomes feasible to less than 10 ms with CRT. The price to pay for more congruences is increased encryption time (by few milliseconds) and ciphertext size (e.g., 126 bytes for three congruences). Note that the performance gains of CRT plateau and a higher number of congruencies do not yield further gains (Table 3). Considering all these parameters, we select (for both standard and sharing modes) two and three congruences for 32- resp. 64-bit values, as a tradeoff between ciphertext expansion and encryption/decryption times.

Memory. Pilatus requires two separate look-up tables (hash tables) for standard mode and data sharing, as we use two different underlying elliptic curves. The former accounts for 67 Mbyte and the latter for 69 Mbyte, with 80-bit security. The tables are shared among all applications served by Pilatus which amortizes the memory requirements among them.

7.1.2 Standard Mode. In the standard mode, we utilize EC-ElGamal to process data encrypted under a single key (no data sharing). While the focus of our benchmark is on EC-ElGamal's performance, we also compare it to the more conventional Paillier cryptosystem, used for instance in CryptDB [46].

Key Generation. Table 1 shows the average setup time on the Nexus 5 device. EC-ElGamal requires less than 0.3 ms to generate keying material with 80-bit security. This is significantly lower than the 1623 ms required for Paillier, even though our key generation additionally includes finding the corresponding primes for the congruences in the CRT. Anyhow, key generation does not frequently occur, compared to encryption and decryption, detailed next.

Ciphertext Size. Figure 10(a) shows the ciphertext sizes for different integer sizes and security levels. In the standard mode, we support 16, 32 and 64-bit integers. Each integer size requires a different number of congruences, leading to a ciphertext size between 42 and 126 bytes in the 80-bit security case. In contrast, Paillier requires 256 bytes, regardless of the integer size. The difference is even more pronounced as the security level increases, negatively impacting network bandwidth and cloud storage.

Encryption/Decryption. Figure 10(b) shows the encryption and decryption time with EC-ElGamal vs. Pailler, for different integer sizes, with both 80-bit and 128-bit security.

[5]Guava: Google Core Lib for Java: https://github.com/google/guava

Integer size [bits]	Encryption time [ms]					Decryption time [ms]					Ciphertext size [bytes]				
	Talos	Pilatus, #congruences				Talos	Pilatus, #congruences				Talos	Pilatus, #congruences			
		2	3	4	5		2	3	4	5		2	3	4	5
32	1.1	**2.4**	3.3	4.4	5.4	42*	**2.0**	2.9	4.0	5.0	42	84	126	164	210
64	1.2	2.5	**3.5**	4.7	5.7	infeasible	139	**6.4**	4.2	5.0					

Table 3: CRT optimization. Performance of Pilatus (EC-ElGamal with CRT) on Nexus 5 with 80-bit security, compared to Talos (i.e., no CRT). *The reported 190 ms in Talos [53] is reduced here to 42 ms with multithreading.

Paillier has constant computation times due to its large padding, while EC-ElGamal sees its performance increase for smaller plaintext. EC-ElGamal outperforms Paillier for encryption with a factor of 3 and more. For decryption, EC-ElGamal is the fastest in all settings but the case of 80-bit security and 64-bit integers. Paillier's sharp decrease in performance with 128-bit security is due to larger key sizes (from 1024 to 3072-bit) and the resulting big number operations.

Figure 10(c) shows the batch decryption time at the mobile device. Even for 64-bit integers, hundreds of items can be decrypted in some hundred milliseconds. This is an important factor for the responsiveness of smartphone applications.

Note that the good performance of EC-ElGamal lies to a large part in our CRT optimization, as discussed in §7.1.1. Moreover, the benefits of an efficient encryption are particularly important in an IoT context, since IoT applications tend to encrypt more data items than they decrypt (all measurements are stored in the cloud, only a subset or aggregates are downloaded for display).

Homomorphic Addition. We measure the homomorphic addition in isolation on a MacBook Pro, thus exclude the bootstrapping overhead of the database's UDF. The homomorphic addition in the standard mode requires between 27 and 82 μs for 16 and 64-bit integers, respectively (see Table 2). This is higher than Paillier (8 μs), which is mainly due to the underlying structure of EC-ElGamal where the ciphertext consists of two EC points. Note that parallelizing the homomorphic addition on the cloud would potentially result in considerable performance gain. In §7.2, we discuss the impact of this overhead on `sum` queries.

7.1.3 Data Sharing Mode. We evaluate the data sharing mode, where the client encrypts data and issues a token such that the cloud can re-encrypt a ciphertext to the target user or group. The key setup is with 4 ms similarly efficient to EC-ElGamal (see Table 1).

Ciphertext Size. Note that we use the same number of congruences as in standard mode. Since our re-encryption scheme is based on bilinear maps, we have to select the parameters such that we achieve at least 80-bit (i.e., subgroup size 160 and extension field size 1024) or 128-bit security. This results in larger ciphertext sizes compared to standard mode (i.e., between 186 and 558 bytes as depicted in Figure 11(a)). This is four times larger than in standard mode, but still comparable with Paillier. After sharing (i.e., re-encryption) the ciphertext sizes expand by a factor of 1.7 due to pairing.

Encryption/Decryption. Figure 11(b) shows the performance of encryption and decryption in the sharing mode, for different integer sizes and security levels. Overall, the sharing mode is slower than standard mode by a factor of 2.2 to 3.3, but remains within acceptable bounds, that is, below 30 ms.

Note that this overhead is, to some extent, offset by the performance gains of offloading the re-encryption operation to the cloud. To enable sharing, the client only needs to generate a token (takes 2.5 ms) and then encrypt the data in sharing mode. As depicted in Figure 11(c), a client can issue a few hundred re-encryption tokens within 500 ms. Issuing a large number of re-encryption tokens becomes relevant after access revocation, as discussed in §5.4. The performance of re-encryption at the cloud is evaluated in §7.2.

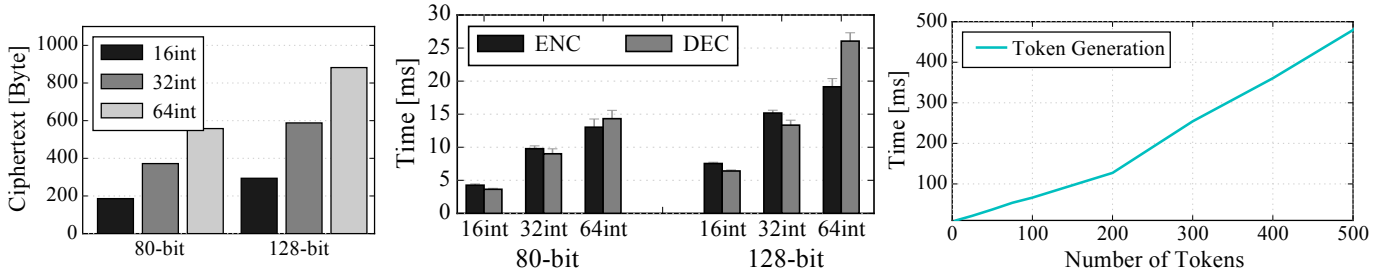
Homomorphic Addition. The homomorphic addition with the data sharing mode is more efficient than the standard mode. Note that with data sharing, we have two types of additions: prior share and post share, which amount to 31 and 15 μs per CRT, respectively (see Table 2).

7.1.4 Constrained IoT Devices. We now turn our attention to constrained IoT devices and evaluate encryption performance with the OpenMote's hardware accelerator. For the time and energy measurements, we rely on a dedicated hardware timer with an accuracy of 1 μs and a mixed signal oscilloscope, respectively. Paillier in 80-bit security requires 1.8 s to encrypt a 16-bit value. The same encryption with EC-ElGamal is significantly more efficient with 126 ms (corresponds to 6.85 mJ). Table 2 compares the results of constrained devices to smartphones. Although the encryption time on constrained devices is more than one order of magnitude higher than encryption on the more powerful IoT gateways, it is still feasible with only 10% of the daily energy budget of a typical Fitbit (400 mAh lithium-polymer battery) to encrypt at a rate of 0.24 Hz. To put this number into context, assuming heart rate tracking at 1 Hz during sport and six times per hour otherwise[6], we can encrypt the heart-rate of a person with 6 hours of sports activity per day.

7.2 System Benchmark

This section evaluates Pilatus as a full system: processing throughput, end-to-end latency, and our two case study applications: Fitbit and Ava.

[6]Microsoft Band: http://www.windowscentral.com/how-often-microsoft-band-checks-your-heart-rate



(a) Ciphertext size (bandwidth/memory).

(b) Encryption/decryption in sharing mode.

(c) Batch token generation (for re-encryption) with 80-bit security and 4x threads.

Figure 11: On-device Data Sharing evaluation. The Client engine performs encryptions and decryptions of data. After revocation, new re-encryption tokens are issued for valid sharing relationships. Note batch encryption/decryption with multithreading (e.g., 4 threads) yields a factor of 3 performance improvement, e.g., decryption of 64-bit integers is reduced to 5.5 and 10 ms, for 80-bit and 128-bit security, respectively.

Methodology. We utilize our client and cloud engines for the system benchmark. The client engine has an average ping time to the Amazon cloud of 22 ms, or it can be co-located at the cloud, neglecting network latency in favor of more isolated throughput measurements. Similar to micro-benchmarks, we rely on the *Stopwatch* class for the time measurements, instrumented at the client engine. To compute the system's throughput at the cloud engine, we initiate SQL `sum` queries over a varying number of values from our client module. For the end-to-end evaluation, the ciphertexts are additionally decrypted and provided to the corresponding application.

Encrypted Query Processing. Figure 12(a) and Figure 12(b) depict the performance of the cloud engine on AWS when performing `sum` SQL queries, over either plaintext or encrypted data. We create queries with variable lengths (i.e., summands) and measure the time to process each, and compute the average system throughput of the cloud engine for a single connection. In Figure 12(a), we initiate the requests locally from the cloud (no network latency) and in Figure 12(b) from the client engine over the Internet.

Without consideration of network communication, plaintext sum operations are about two orders of magnitude faster than the homomorphically encrypted sums. With network communication, the relative performance loss is lower, in particular when only a few items are added and network delay is the bottleneck. However, the larger the number of items, the more the overhead of homomorphic additions impacts the performance, e.g., with 1000 values to be added, the performance loss reaches a factor 2.7. Note that such queries are highly parallelizable, allowing for better performance; plaintext operations already benefit from parallelization. Our current evaluation results show the lower bound performance and in future work, we plan to parallelize our routines at the database.

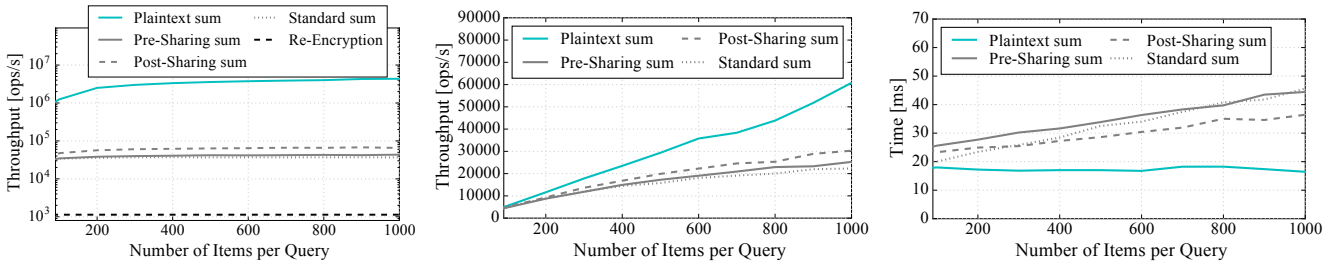
For data sharing, the computations take place offline in the cloud, without user interaction. The throughput of re-encryptions per data item amounts to 1136 re-encryptions per second, with a single thread (see Figure 12(a)). Note that re-encryption is the most expensive operation as it involves expensive pairing. The re-encryptions should also be parallelized in the cloud to reach the best performance.

App (mode)	Upload (encrypt)		View (decrypt)	
	[s]	[# items]	[ms]	[# items]
Fitbit (Standard)	1.3	1500	30	50
Ava (Sharing)	31	9000	127	50

Table 4: Fitbit and Ava enhanced with Pilatus. Encryption overhead when uploading one day worth of data and decryption overhead at visualization for the most costly view. Security is set to 80-bit, all data items are 32-bit values, and multithreading enabled.

End-to-end Latency. Figure 12(c) depicts the end-to-end latency for varying `sum` queries. The latency values follow a similar trend as the throughput values, as depicted in Figure 12(b). For lower range `sum` queries, the average performance of data sharing and standard mode are close to queries over plaintext. For larger ranges, the average latency increases by a factor of 2 and 2.7, respectively. To guarantee a smooth user interaction with encrypted data, the latency should be below 1 s, which is the case even for larger ranges (i.e., below 50 ms for 1000 items).

Applications. Our two Android applicationsrun Pilatus on real-world collected data and allow the user to upload encrypted data and interact with them similarly to the original apps (see Table 4). Our FitBit app adds an overhead of 1.3 s for uploading data of one day – an operation that takes place in the background. While rendering different views of the app (e.g., daily, weekly, detailed graphs), we measure a maximum overhead of 32 ms due to decryptions. Note that we use no local caching to emulate worst case scenarios. Our Ava fertility tracking app collects data from a more diverse set of sensors at a higher granularity during sleep and hence produces more data points. Additionally, we discuss the numbers in the context of the more expensive data sharing mode. Our Ava app induces an overhead of 31 s for uploading one day worth of data. For rendering different views after sharing, the maximum overhead is 127 ms. For both apps, the overhead due to decryption is well below the 1 s requirement. Hence, the user experience with Pilatus remains unaltered.



(a) Operations in the cloud, initiated locally at the cloud. (b) Operations in the cloud, initiated by the client over the network (22 ms ping). (c) End-to-end latency including decryption.

Figure 12: Cloud evaluation of Standard Mode and Data Sharing, with our cloud engine running on Amazon, with 80-bit security. Pre- and post-sharing sum refer to `sum` queries before and after sharing.

Conclusion. Our evaluation shows the practicability of Pilatus, especially for mobile platforms which play a vital role in the IoT ecosystem (i.e., as the gateways). The end-to-end latency results show that Pilatus succeeds in preserving the user experience while interacting with encrypted data. Pilatus induces a moderate overhead for an increased level of data privacy and security.

8 RELATED WORK

In the following, we discuss important research directions in relation to Pilatus.

Encrypted Search. Recent advancements of fully homomorphic encryption [22] have resulted into implementable schemes [14, 23, 50], which are however presently too slow for real world applications. Searchable encryption schemes support only a limited set of operations, but can be efficiently used in specialized domains. Song et al. [58] introduced the first encrypted search scheme for text files, where the metadata is encrypted deterministically and hence searchable. Their idea is based on deterministically encrypting the meta information of files, and hence being able to search over them. Follow-up schemes address other problems such as encrypted data de-duplication [33], deep packet inspection [54], and private network function virtualization [3]. More capable search schemes [12, 46, 49, 50, 53, 55], targeted for structured databases, employ additional techniques such as partially homomorphic and order-preserving encryptions. Among these, CryptDB has early adopters in industry [24, 38]. Monomi [62] improves the performance of CryptDB and extends supported queries. In CryptDB, the application server has access to keys and carries out en-/decryptions. Hence, it can leak information if compromised. Talos [51, 53] tailors CryptDB for IoT devices and eliminates the need to trust the application server. Mylar [47] introduces encrypted text file search with multiple keys. Shi et al. [56] propose private aggregation for time series data, which blends secret sharing with homomorphic encryption. Access patterns to encrypted data still leak sensitive information about the plaintext data. This shortcoming can be addressed with Oblivious RAM approaches [48, 59]. Pilatus is the first practical system to support processing of multi-key encrypted data and is tailored for constrained devices. This opens practical encrypted data processing to a new space of applications that existing systems do not support.

MPC. In traditional Secure Multi-Party Computation (MPC) [66], private functions are computed among a set of users without a trusted party. Hereby individual values from participating users are kept confidential, while the outcome can be public. This requires high interaction between users, which would drain the limited resources of mobile platforms. With the rise of cloud computing, server-aided/outsourced MPC approaches have emerged. However, these schemes are either only of theoretical interest [36] or require at least two non-colluding servers, where for instance one server has only access to encrypted data and the other server has access to the keys [42, 45].

Trusted Computing. An orthogonal approach to encrypted computing assumes a trusted computing module on an untrusted cloud environment [6, 9, 37]. The data remains encrypted at rest and is decrypted in the trusted module for computations. This approach is appealing to data center operators, due to control over hardware. However, it implies that users consider the trusted computing module trustworthy. Autocrypt [61] combines PHE and trusted computing to enable encrypted computing on sensitive Web data on virtual machines.

Re-Encryption. The idea of Re-Encryption (RE) has been initially proposed for email forwarding. The initial schemes [10, 30] have the bi-directional property and are not resistant against collusion. Moreover, the parties need to exchange their private keys. The later schemes [4, 26] address these weaknesses and are uni-directional and non-interactive. Pilatus utilizes the RE scheme by Ateniese et al. [4] to allow transformation of encrypted data for data sharing. More importantly, we extend this scheme with the CRT technique, to render it efficient. The symmetric-key RE based on the key-homomorphic PRF scheme [13] lacks our required homomorphic property and master-secret secrecy. Sieve [65] utilizes this key-homomorphic scheme to provide cryptographically enforced access control for cloud data. Sieve's key revocation assumes that the cloud does not yield access to compromised shared keys. Otherwise, the new key will be automatically compromised as the cloud can use the old key to compute the new key from the re-keying delta. In Pilatus, even with access to the revoked key of Bob and re-keying delta, the cloud cannot learn the new key of Alice.

9 CONCLUSION

We presented Pilatus, a new practical system tailored for the IoT ecosystem. We empower the user with full control over their data, despite it being stored in third-party clouds. In Pilatus, the cloud does not have access to any secret keys and stores only encrypted data. It can though process queries on encrypted data and re-encrypt it for sharing. Our sharing scheme comes with cryptographic guarantees and the possibility of revocation. We have optimized the underlying cryptographic operations towards mobile platforms. Our implementation and case studies on Fitbit and Ava show that Pilatus has reasonable overhead in processing time and end-to-end latency. We anticipate the presented cryptosystem and open-source platform to be helpful for the design of secure mobile applications and to enable further research in this field.

10 ACKNOWLEDGMENTS

The authors would like to express their gratitude to David Wu, Philip Levis, and Srdjan Capkun for the insightful discussions during the design phase of Pilatus. We thank Friedemann Mattern and Alexander Viand for their comments on earlier versions of this paper. Moreover, we are thankful to the anonymous reviewers and our shepherd, Xiuzhen Cheng, for their valuable feedback. This work was partly supported by a grant from CPER Nord-Pas-de-Calais/FEDER DATA and VINNOVA, Sweden's innovation agency.

REFERENCES

[1] Diego F. Aranha and Conrado P.L. Gouvêa. 2017. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic. (2017).
[2] Diego F. Aranha, Koray Karabina, Patrick Longa, Catherine H. Gebotys, and Julio Lopez. 2011. Faster Explicit Formulas for Computing Pairings over Ordinary Curves. In *EUROCRYPT* .
[3] Hassan Jameel Asghar, Luca Melis, Cyril Soldani, Emiliano De Cristofaro, Mohamed Ali Kaafar, and Laurent Mathy. 2016. SplitBox: Toward Efficient Private Network Function Virtualization. In *Workshop on HotMIddlebox*.
[4] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. 2005. Improved Proxy Re-encryption Schemes with Applications to Secure Distributed Storage. In *NDSS*.
[5] Ava. 2016. Fertility Tracking Bracelet. Online: avawomen.com. (2016).
[6] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. In *ACM SIGMOD*.
[7] Mario Ballano Barcena, Candid Wueest, and Hon Lau. 2014. *How safe is your quantified self?* Technical Report. Symantec.
[8] Paulo Barreto and Michael Naehrig. 2005. Pairing-Friendly Elliptic Curves of Prime Order. In *international Conference on Selected Areas in Cryptography (SAC)*. 319–331.
[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *USENIX OSDI*.
[10] Matt Blaze, Gerrit Bleumer, and Martin Strauss. 1998. Divertible Protocols and Atomic Proxy Cryptography. In *EUROCRYPT*.
[11] Dan Boneh and Matthew K. Franklin. 2001. Identity-Based Encryption from the Weil Pairing. In *CRYPTO*.
[12] Dan Boneh, Craig Gentry, Shai Halevi, Frank Wang, and David J. Wu. 2013. Private Database Queries Using Somewhat Homomorphic Encryption. In *Applied Cryptography and Network Security (ACNS)*.
[13] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. 2013. Key Homomorphic PRFs and Their Applications. In *CRYPTO*.
[14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In *Innovations in Theoretical Computer Science Conference*.
[15] Charles L. Buxton and William B. Atkinson. 1948. Hormonal Factors Involved in the Regulation of Basal Body Temperature During the Menstrual Cycle and Pregnancy. *The Journal of Clinical Endocrinology & Metabolism* 8, 7 (1948), 544–549.
[16] Cunsheng Ding, Dingyi Pei, and Arto Salomaa. 1996. *Chinese remainder theorem: applications in computing, coding, cryptography.* World Scientific.
[17] Stuart Dredge. 2013. Yes, those Free Health Apps are Sharing your Data with other Companies. Guardian, Online: theguardian.com/technology/appsblog/2013/sep/03/fitness-health-apps-sharing-data-insurance. (2013).
[18] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. 2004. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *IEEE LCN*.
[19] Economist. 2015. Things are looking app. Online: economist.com/news/business/21694523-mobile-health-apps-are-becoming-more-capable-and-potentially-rather-useful-things-are-looking. (2015).
[20] Experian. 2015. Data Breach Industry Forecast. *White Paper* (2015).
[21] David Freeman, Michael Scott, and Edlyn Teske. 2010. A Taxonomy of Pairing-Friendly Elliptic Curves. *Journal of Cryptology* 23, 2 (2010), 224–280.
[22] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *ACM Symposium on Theory of Computing (STOC)*.
[23] Craig Gentry and Shai Halevi. 2011. Implementing Gentry's Fully-Homomorphic Encryption Scheme. In *EUROCRYPT*.
[24] Google. 2015. Encrypted Bigquery Client. Online: github.com/google/encrypted-bigquery-client. (2015).
[25] Google. 2015. Google Sign-In. Online: developers.google.com/identity/sign-in/android/. (2015).
[26] Matthew Green and Giuseppe Ateniese. 2007. Identity-Based Proxy Re-encryption. In *Applied Cryptography and Network Security (ACNS)*.
[27] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Song. 2014. ShadowCrypt: Encrypted Web Applications for Everyone. In *CCS*.
[28] Yin Hu, William J. Martin, and Berk Sunar. 2012. Enhanced Flexibility for Homomorphic Encryption Schemes via CRT. In *Applied Cryptography and Network Security (ACNS)*.
[29] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *NDSS*.
[30] Anca Ivan and Yevgeniy Dodis. 2003. Proxy Cryptography Revisited. In *NDSS*.
[31] Stephanie Jernigan, Sam Ransbotham, and David Kiron. 2016. Data Sharing and Analytics Drive Success With IoT. *MIT Sloan Management Review* (September 2016).
[32] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. 2016. Verena: End-to-End Integrity Protection for Web Applications. In *IEEE Symposium on Security and Privacy*.
[33] Sriram Keelveedhi, Mihir Bellare, and Thomas Ristenpart. 2013. DupLESS: Server-Aided Encryption for Deduplicated Storage. In *USENIX Security*.
[34] Keybase. 2016. Publicly Auditable Proofs of Identity. Online: keybase.io. (2016).
[35] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. 2013. OAuth 2.0 Threat Model and Security Considerations. *IETF, RFC 6819* (January 2013).
[36] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. 2012. On-the-fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption. In *ACM STOC*.
[37] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*.
[38] Microsoft. 2015. Always Encrypted (Database Engine). Online: msdn.microsoft.com/en-us/library/mt163865.aspx. (2015).
[39] Victor S. Miller. 2004. The Weil pairing, and its efficient calculation. *Journal of Cryptology* 17, 4 (2004), 235–261.
[40] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *CCS*.
[41] Jakob Nielsen. 1993. *Usability Engineering*. Morgan Kaufmann Publ.
[42] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. 2013. Privacy-Preserving Ridge Regression on Hundreds of Millions of Records. In *IEEE Symposium on Security and Privacy*.
[43] Parmy Olson. 2014. For Google Fit, Your Health Data Could Be Lucrative. Forbes, Online: forbes.com/sites/parmyolson/2014/06/26/google-fit-health-data-lucrative. (2014).
[44] Pascal Paillier. 1999. Public-key Cryptosystems Based on Composite Degree Residuosity Classes.. In *EUROCRYPT*.
[45] Andreas Peter, Erik Tews, and Stefan Katzenbeisser. 2013. Efficiently Outsourcing Multiparty Computation under Multiple Keys. *IEEE Transactions on Information Forensics and Security* 8, 12 (2013), 2046–2058.
[46] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *ACM SOSP*.
[47] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. 2014. Building Web Applications on Top of Encrypted Data Using Mylar. In *USENIX NSDI*.
[48] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security*.
[49] Tahmineh Sanamrad, Lucas Braun, Donald Kossmann, and Ramarathnam Venkatesan. 2014. Randomly Partitioned Encryption for Cloud Databases. In *DBSec*.
[50] Hossein Shafagh. 2015. Toward Computing Over Encrypted Data in IoT Systems. In *XRDS: Crossroads, The ACM Magazine for Students Volume 22 Issue 2, Winter 2015*. 48–52.

[51] Hossein Shafagh, Lukas Burkhalter, and Anwar Hithnawi. 2016. Demo Abstract: Talos a Platform for Processing Encrypted IoT Data. In *ACM SenSys*.
[52] Hossein Shafagh and Anwar Hithnawi. 2017. Privacy-preserving Quantified Self: Secure Sharing and Processing of Encrypted Small Data. In *ACM Workshop on Mobility in the Evolving Internet Architecture (MobiArch)*.
[53] Hossein Shafagh, Anwar Hithnawi, Andreas Dröscher, Simon Duquennoy, and Wen Hu. 2015. Talos: Encrypted Query Processing for the Internet of Things. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*.
[54] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2015. Blind-Box: Deep Packet Inspection over Encrypted Traffic. In *ACM SIGCOMM*.
[55] Elaine Shi, John Bethencourt, T.-H.H. Chan, Dawn Song, and Adrian Perrig. 2007. Multi-Dimensional Range Query over Encrypted Data. In *IEEE Symposium on Security and Privacy*.
[56] Elaine Shi, Richard Chow, T-H. Hubert Chan, Dawn Song, and Eleanor Rieffel. 2011. Privacy-preserving Aggregation of Time-series Data. In *NDSS*.
[57] Nigel Paul Smart. 2003. *Cryptography: an Introduction*. Vol. 5. McGraw-Hill New York.
[58] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *IEEE Security and Privacy*.
[59] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS*.
[60] Texas Instruments. 2013. CC2538 System-on-Chip for 2.4-GHz IEEE 802.15.4. Online: www.ti.com.cn/cn/lit/ug/swru319c/swru319c.pdf. (2013).
[61] Shruti Tople, Shweta Shinde, Zhaofeng Chen, and Prateek Saxena. 2013. AUTOCRYPT: Enabling Homomorphic Computation on Servers to Protect Sensitive Web Content. In *CCS*.
[62] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing Analytical Queries Over Encrypted Data. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*.
[63] Frederik Vercauteren. 2010. Optimal Pairings. *IEEE Transactions on Information Theory* 56, 1 (2010), 455–461.
[64] Boyang Wang, Ming Li, Sherman S. M. Chow, and Hui L. 2014. A Tale of Two Clouds: Computing on Data Encrypted under Multiple Keys. In *IEEE Conference on Communications and Network Security*.
[65] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. 2016. Sieve: Cryptographically Enforced Access Control for User Data in Untrusted Clouds. In *USENIX NSDI*.
[66] Andrew C. Yao. 1982. Protocols for Secure Computations. In *Symposium on Foundations of Computer Science*. 160 – 164.