P^2-SWAN: Real-time Privacy Preserving Computation for IoT Ecosystems

Marc X. Makkes, Alexandru Uta, Roshan Bharath Das, Vladimir N. Bozdog, Henri Bal
Dept. of Computer Science, Vrije Universiteit Amsterdam, The Netherlands
{m.x.makkes, a.uta, r.bharathdas, n.v.bozdog, h.e.bal}@vu.nl

***Abstract*—Sensitive personal user-data collected by Internet-of-Things (IoT) devices is vulnerable to information leaks when uploaded to third-party cloud computing infrastructures. Even though data is encrypted before being sent, to perform analyses on the received data, the computing infrastructure typically decrypts the data, and then performs computation. Therefore, during computation, data can be leaked by means of *honest-but-curious* system administrators. To overcome this vulnerability, homomorphic encryption enables "blind" computation directly on encrypted data, thus rendering obsolete any data leaks. However, homomorphic encryption is highly resource demanding, as it performs many compute-intensive operations during encryption, while also increasing the size of ciphertexts, which makes it unsuitable for low-powered (mobile) IoT devices. For similar reasons, performing operations on encrypted data is also a challenging task, especially when real-time decision-making is needed. In such scenarios, efficient solutions must be augmented by placing computation close to the data: at the network edge.**

In this paper, we introduce *privacy preserving* SWAN (P^2-SWAN), a homomorphic-encryption enabled mobile computing framework. Even though such encryption adds significant computational overhead, our evaluation shows that it is feasible on low-powered (mobile) devices. The overhead induced on such devices is minimized due to our carefully crafted implementation. We show that performing encrypted operations achieves excellent scalability, thus only modest numbers of computing servers can handle the load for data generated by millions of devices. Furthermore, our proposed approach achieves real-time computation not only for encrypting data on mobile devices, but also for performing encrypted computation.

I. Introduction

The field of distributed sensing and monitoring is rapidly evolving. Recent estimates [1] for the number of *Internet of Things* (IoT) [2] devices foresee that by the year 2020 there will be up to 50 billion devices connected to the Internet. Flexible solutions for processing IoT data include cloud or fog [3] computing. Such solutions provide interfaces for storing or performing various real-time analyses for sensor-generated data. Common examples of such analyses include traffic monitoring, air-quality measurements, etc. The aforementioned applications do not handle sensitive data, thus the communication (or even the computation) need not necessarily be encrypted.

Recently, an abundance of sensors in smartphones (e.g., GPS, step-count, sound) or wearable devices (e.g., heart-rate, accelerometer) have become available to customers for keeping track of various personal data (e.g., tracking sleep, physical activity, last known location). The default behavior of such devices is that the personal user data is uploaded to (third-party) cloud infrastructures for storage and possibly for (health) monitoring and advice on how to improve fitness levels. Irrespective of the placement scheme, either on the wearable device vendor's private infrastructure or a third-party facility, the data generated by such devices is extremely sensitive and should be only available to the owner and trusted entities (i.e., the personal doctor).

The downside with either private or public data processing infrastructures is that data owners need to trust third-party platforms or administrators. In addition, in typical IoT solutions, such as ThingSpeak [4], encryption is only provided between IoT devices and the data collector. Whenever data analysis is performed (i.e., when the user requests advice on how to train better), the data needs to be decrypted. Such data storing and processing platforms are susceptible to data leaks by means of curious administrators who can access the system and infer knowledge from the private customer data. Therefore, we argue that encryption alone is not enough for user privacy: the computation itself needs to be performed on encrypted data (which only the user or the doctor can later decrypt).

To solve this problem, and allow computation on encrypted data, the solution is *homomorphic encryption* [5]. This blind computation model, i.e., doing operations on encrypted data without revealing its contents, is seen as the holy grail [6] for off-loading operations to possibly untrusted infrastructures, as nothing can be inferred about what is being computed (upon). Fully homomorphic encryption [7], [8] is not feasible for current computing systems [9] as its complexity renders it impractical. Gentry et. al. [10] already presented a first approach at implementing fully homomorphic encryption. Their results range from 30 seconds to 30 minutes for a single encryption, while the public key size increases from 70 MB to 2.3 GB. Such a scheme is clearly impractical even for state-of-the-art processors, let alone low-powered IoT devices, such as smartphones or sensor boards.

However, for the IoT domain, when restricting the possible input space (i.e., to integers) and the applicable operations (i.e., to simple arithmetic operators: addition, multiplication, exponentiation), the problem becomes tractable. Such a scheme can be efficiently implemented using the Paillier cryptosystem [11]. Using our optimized implementation [12] of the Paillier cryptosystem, in this paper we study the feasibility of achieving "blind computation" for IoT-generated data in **real-time** decision making scenarios.

Computation on encrypted data has already been proposed for securing database systems [13], or large-scale stream processing systems [14]. Such approaches have been designed for industrial-scale, power-hungry computing platforms (clouds, clusters). However, when securing the computation on data generated by small, low-powered IoT devices, implementing such a system becomes more challenging.

First, such devices have much less computational capacity than a fully-fledged server and are battery-powered. Therefore, the encryption becomes an expensive operation, not only because it requires additional computation, but also due to its increased power consumption. Second, there are potentially many (i.e., millions) such devices, each sending messages at a possibly high frequency. As doing computation on encrypted data requires more CPU cycles on the server machines, implementing an efficient computation engine, that scales to possibly millions of clients also becomes a difficult challenge. Third, in IoT it is often important to perform real-time analysis on sensor data (i.e., patient monitoring, disaster management). Therefore, not only the encryption performed on sensors needs to have a fast response time, but also the data analysis. For this paper, we consider *real-time processing* to occur at a greater (or equal) rate than the *data generation rate*. Therefore, **real-time encryption** is achieved when the encryption rate is greater (or equal) than the sensor reading rate. Conversely, **real-time data analysis** is achieved when the processing system analyzes data at a greater (or equal) throughput than the incoming data throughput.

Even if efficient computation is achieved, an important component of real-time decision-making is minimizing network delays. Therefore, the data analysis platform should be placed as close as possible to the data-generating devices, i.e., in an **edge** cloud infrastructure. Our proposed solution can be applied even in setups where connectivity is restricted to only mobile (4G) networks. By placing computing capacity at the edge, i.e., on the cell towers, data generated by mobile devices can be quickly computed upon, as depicted in Figure 1.

To address these challenges, in this paper, we present P^2-SWAN, a framework for performing real-time edge secure computation in third-party untrusted datacenters, while minimizing the impact of the cryptographic system on both the sensor nodes and the computation engine. Our implementation is an extension of our *Sensing With Android Nodes* (SWAN) [15]–[17] platform.

Our contributions are as follows:

- We show that Paillier homomorphic encryption is feasible on mobile devices;
- We show that the impact of homomorphic encryption on the device battery draining is minimal;
- We show that our system achieves real-time computation, both on mobile devices and edge cloud platforms;
- We show that the performance overhead of the encryption operation is negligible, while the data processing engine achieves excellent scalability, even when compared to large-scale encrypted stream computation platforms, such as [14];
- We extend SWAN to include homomorphic encryption and encrypted computation on sensor-generated data.

The remainder of this paper is organized as follows. In Section II we present and discuss background knowledge and related work. In Section III we discuss the design of P^2-SWAN, while Section IV presents our use-cases and supported homomorphic operations. The experimental evaluation of P^2-SWAN is presented and discussed in Section V, while Section VI draws the conclusions.

II. Background and Related Work

This section discusses the background knowledge on top of which P^2-SWAN is built. We also present related approaches to our work.

A. Related Work

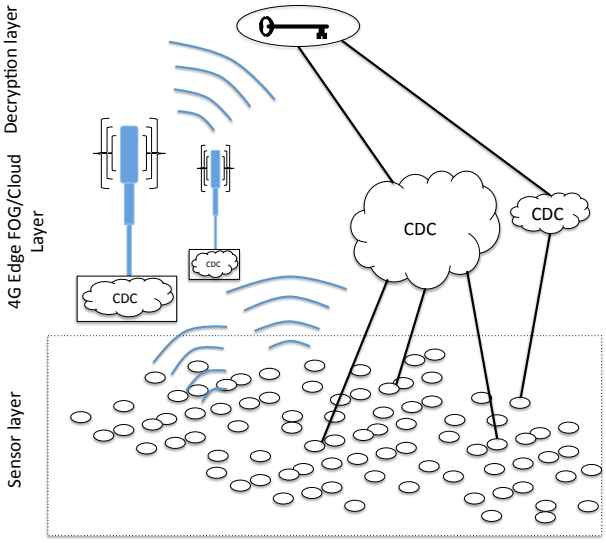


Fig. 1. A general edge computing scenario. Many mobile devices with sensors send sensor information over 4G to a cell-tower with a cloud data center (CDC) or directly to a cloud data center by a wired connection. The cloud data centers can run on-demand virtual machines to process sensor information. Once completed, the results are sent to a device that holds the private key.

Off-loading millions of sensors which produce a manifold of data points per second, requires high performance implementation of protocols and careful planning on where to process which part of the application. It is even more challenging to do this in a privacy preserving way. In the past decade several researchers have dealt with various security issues in low-energy ad hoc wireless networks [18], wireless sensor networks [19], as well as IoT devices [20].

One approach is to securely forward the sensor information by the networks itself. Wireless sensor networks are networks in which nodes work cooperatively to pass their data through the network to specific end-points. Using homomorphic encryption data aggregation is securely possible [21]. In this scenario, a tree structure is created with one or more data collection points. Every node in the network aggregated information from its children and send to its parents. This model

usually creates an unbalanced processing in the network, and some nodes process more than others. In addition, the network size impacts the latency when information arrives at a collection point, as every low power node collects, processes and sends new packets. Also, depending on the type of tree chosen in the network, fat trees may lead to bottlenecks in the networks. In our approach we allocate one or more virtual machines to process the data securely, lifting the computation burden from intermediate nodes, which has a lower impact on the battery life of all nodes.

Another approach is to store sensor data in a secure database, i.e., storing the encrypted values of data points, and run special queries over them to retrieve data. There are several solutions for storing and querying data securely, such as StreamForce [22] and Cryptdb [13]. These systems rely on cryptography to securely outsource data to an untrusted server or virtual machines running in cloud data centers. As opposed to StreamForce, Cryptdb allows to perform operations on single data entries, which are encrypted, using a homomorphic cryptosystem. These systems are more targeted at querying, i.e., less frequently changing data, as opposed to data stream processing. Data stream processing systems, such as the recently introduced Polystream [14], are designed for high volume, process and forget type of applications. The main focus of Polystream is access control to the data streams while here we focus on performance. Moreover, our model is different, we assume there is only one destination, i.e., the node which has the private key. Both Cryptdb and Polystream use 1024bit keys, which is strongly discouraged by [23]–[25]. All recommendations aim for at least 2048-bit encryption lower-bound security.

B. SWAN

We base our work upon SWAN [16], which is a framework for easily building context-aware applications for Android smart-phones. SWAN acts as a middleware between applications and sensors. It focuses on helping application developers by providing a high-level abstraction for accessing the sensors. It also eliminates redundancy caused by multiple sensor-based applications running in parallel by providing a centralized solution for collecting and storing sensor data locally. We also provide a flexible solution called SWAN-Fly [17], to send the sensor data to the cloud of the application developers' choice.

Multiple applications can interact with SWAN using its domain specific language called SWAN-song [15]. The application registers a SWAN-song expression to SWAN, which evaluates the expression and sends back the result.

As an example, the following SWAN-song expression gets the current light intensity:

```
self@light:lux{MAX,1000ms}
```

where $self$ is the location of the sensor, $light$ is the sensor identifier, lux is the value path (sensor property that has to be read), MAX represents the history reduction mode and $1000ms$ represents the history window. This expression will compute the maximum of the values generated by the light sensor over the last 1000 milliseconds.

The core of SWAN is the expression evaluation service that is responsible for querying the sensors and evaluating the registered expressions whenever new sensor readings are available. It also communicates with the caller applications, which are notified upon successful evaluation of expressions.

Sensor data is gathered in SWAN by means of Android services that query various resources, such as hardware sensors (on-device sensors like GPS, accelerometer, light, battery or sensors in external devices like heart rate monitor). All sensor services run in separate processes, so if SWAN crashes for any reason, the sensor services will remain stable while the affected components recover. This prevents data from being lost. The communication between sensor services and the other components is performed using inter-process communication.

C. Cryptographic Primitives

Security and privacy are often a great concern when dealing with sensitive data. To protect against snooping adversaries, encryption facilitates privacy. Many implementations [19], [26] that do in-network aggregation or dedicated aggregation nodes assume that all nodes are trusted, even if these are virtual machines provide by a third party cloud provider. This is a potential problem, because the sensor data processed by the in-network aggregation is performed in the clear, the message is decrypted prior to aggregation. For some applications, such as open data, this might be acceptable. In other cases, such as medical sensor data, this is definitely not acceptable. Symmetric-key encryption, such as AES [27] provides security, but when using it in-network or dedicated aggregators need to decrypt, aggregate, and finally encrypt before sending the information to the end node. Homomorphic encryption can be used instead of symmetric-key encryption to overcome these problems.

Homomorphic encryption allows operations on encrypted data. Such operations are then reflected in the plaintext. This implies that nodes at the data aggregation layer are unable to see the data in the clear. Rather, they perform operations on ciphertexts. Most homomorphic cryptosystems are non deterministic, i.e., for encryption of a plaintext m there are many possible ciphertexts. This also means that the plaintext is shorter than the resulting ciphertext. This property makes the homomorphic encryption schemes resistant against dictionary attacks. In this paper we implement the Paillier homomorphic cryptosystem [11]. The Paillier cryptosystem is an additive homomorphic system, which allows additions and multiplications on ciphertexts. The Paillier cryptosystem has two variants: the main, and the subgroup variant. The notable difference is that the private key of the subgroup variant is smaller, which allows for faster decryption. In this paper we implement the subgroup variant using a smaller decryption key while having the same security. The subgroup variant works as follows:

Key Generation: Let N be a RSA modulus $N = p \cdot q$, where p and q are large prime integers. Let $\lambda = lcm(p-1, q-1)$

and choose α such that it divides λ. Let $h \in \mathbb{Z}^*_{N^2}$ such that it has maximal order of αN, and $g \equiv h^{\frac{\lambda}{\alpha}} \pmod{N^2}$. The public key is (g, N), and the private key is α

Encryption: To encrypt a message $m \in \mathbb{Z}^*_N$, we randomly choose $r \in \mathbb{Z}^*_N$ and compute the ciphertext $c \equiv g^{m+rN} \pmod{N^2}$.

Decryption: The decryption of c is defined as $\frac{L(c^\alpha \pmod{N^2})}{L(g^\alpha \pmod{N^2})} \pmod{N}$. The $L(u)$ function is defined as $\frac{u-1}{N}$ and takes input of $S_N = \{u < N^2 | u \equiv 1 \pmod{N}\}$

The Paillier cryptosystem holds the following additive homomorphic properties:

$$\forall m_1, m_2 \in \mathbb{Z} \text{ and } k \in \mathbb{N}$$

$$\left.\begin{array}{r} D(E(m_1) \cdot E(m_2) \pmod{N^2}) \\ D(E(m_2) \cdot g^{m_2} \pmod{N^2}) \end{array}\right\} = m_1 + m_2 \pmod{N}$$

$$\left.\begin{array}{r} D(E(m_1)^{m_2} \pmod{N^2}) \\ D(E(m_2)^{m_1} \pmod{N^2}) \end{array}\right\} = m_1 \cdot m_2 \pmod{N}$$

$$D(E(M_1)^k \pmod{N^2}) = km \pmod{N}$$

III. Design

A. Architecture

In this section we describe the security model of our introduced system as well as the threat model. The scenario from Figure 1 presents our working assumptions and overall system model. P^2-SWAN is composed of three layers (Figure 2) 1) sensing nodes generate data and forward it to the Cloud/Fog computation layer; 2) the data processing engine where the operations are performed on the sensor data; 3) the result is forwarded to the device with the private key, which can decrypt the result. Here we assume that the decryption device is a low power device, such as a smartphone or a sensor node.

The main goal of P^2-SWAN is to handle a scenario where information processing can be securely offloaded to an untrusted middle layer. Hence, P^2-SWAN guarantees the confidentiality of data that flows from the sensing nodes to the decryption devices. Although P^2-SWAN protects data confidentiality, it does not ensure the integrity, or computation completeness of the results returned by the data processing engine. For example, P^2-SWAN does not protect against an adversary that could compromise the system, which runs the data processing machine and can modify the cipher texts (by applying new homomorphic operations). Another example is a malicious administrator that stops all processes and connections, effectively halting the data processing. We continue by describing the threat model and security guarantees addressed by P^2-SWAN under those threat models.

B. Threat: System Compromise

In this threat model, the data processing engine protects against a curious system administrator or external adversary with full access to the system. Here, the goal is to have data confidentiality, but not integrity or availability. The adversary is assumed to be passive and to want to learn the contents of confidential data. In addition, the adversary does not perform any changes to protocols and processing of data. However, the adversary has full access to the physical machine, software, memory contents and applications running on the system. This is also known as the semi-honest model [28]. Here, we assume that the adversary has full control over the untrusted edge datacenter system but not over the P^2-SWAN devices. This threat model is increasingly important as the use of public cloud resources and outsourcing is growing.

C. Approach and Guarantees

P^2-SWAN aims to protect data confidentiality a against semi-honest adversary by processing the data encrypted with homomorphic encryption. P^2-SWAN devices first encrypt sensor data prior to sending it to the data processing engine. Our approach is to allow the data processing engine to perform processing on encrypted data as it would on unencrypted data, by enabling it to compute certain functions, such as addition and multiplication. For example, one can compute the arithmetic mean $\frac{1}{n}\sum_{i=1}^{n} a_i$ of all n sensor values, in which the sum, s is processed homomorphically. The produced value s together with n is sent to a device that holds the private key. The device then decrypts s and multiplies the result with $\frac{1}{n}$. This result produces the arithmetic mean. The only information that the data processing engine reveals is the number of sensor data points, not its contents. In addition, the data processing engine does not hold private keys, and uses the public value of N^2 to processes data. Hence, this guarantees data confidentiality.

D. Implementation

*1) **Mobile Devices - Encryption Manager**:* P^2-SWAN in the mobile device is an extension of our SWAN framework. Mobile applications can register expressions to SWAN to gather various sensor data. The extended SWAN-Song language [17] allows the developer to easily create sensor based expressions that can be used to send the sensor data to the edge cloud. The sensor data gathered can be raw (e.g., current sound level) or processed (e.g., average sound level data over a period of 5 minutes). The evaluation engine is responsible for processing the sensor data. After processing, it sends the data to the encryption manager. The encryption manager supports both AES and Paillier cryptosystems. The AES encryption is enabled using a standard cypher library. In case of homomorphic encryption, we use the Paillier cryptosystem.

To improve the Pailier cryptosystem on mobile devices, we apply the following two techniques from our earlier work [12]: pre-computation and simultaneous multi-exponentiation. Using these techniques we reduce the encryption of message m with random number r from $g^{m+rN} \pmod{N^2}$ to $g^m(g^N)^r \pmod{N^2}$ with g^N pre-computed. This transforms the computation problem from an exponentiation with a large number of bits, i.e., rN to a multi-exponentiation $g^m \cdot (g^N)^r$, with fewer bits. Here we pre-compute a table with 2^{2k} entries holding different pre-computed exponentiations, see Table I.

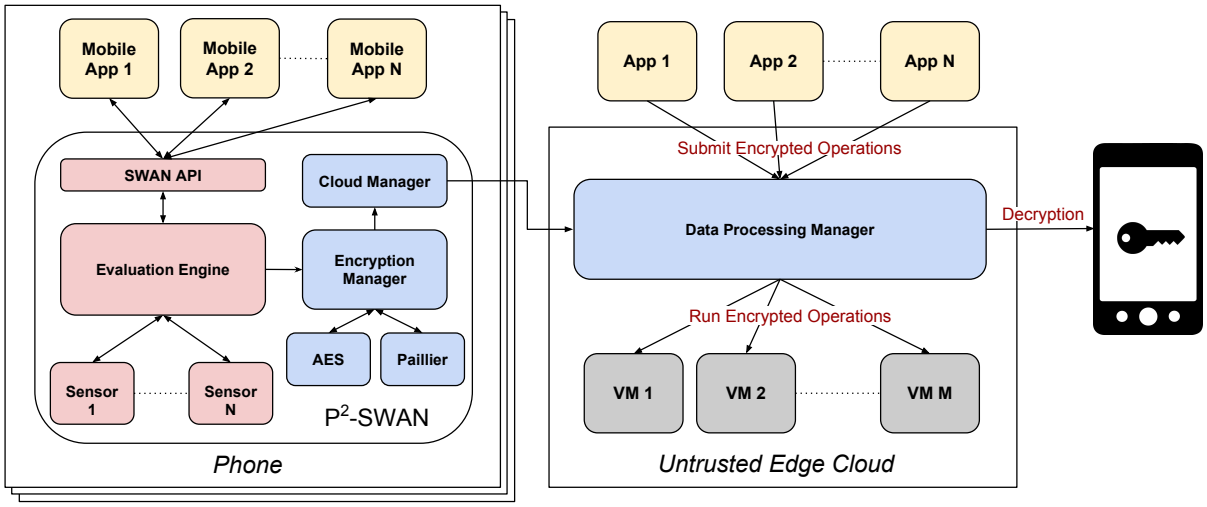


Fig. 2. P^2-SWAN Architecture and Functionality.

TABLE I
A PRE-COMPUTED TABLE OF $2^k \times 2^k$. ALL ENTRIES ARE $\pmod{N^2}$

1	g	g^2	$\cdots$	g^{2^k-1}
(g^N)	$g \cdot g^N$	$g^2 \cdot (g^N)$	$\cdots$	$g^{2^k-1} \cdot (g^N)$
$(g^N)^2$	$g \cdot (g^N)^2$	$g^2 \cdot (g^N)^2$	$\cdots$	$g^{2^k-1} \cdot (g^N)^2$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$(g^N)^{2^k-1}$	$g \cdot (g^N)^{2^k-1}$	$g^2 \cdot (g^N)^{2^k-1}$	$\cdots$	$g^{2^k-1} \cdot (g^N)^{2^k-1}$

To compute the simultaneous multi-exponentiation, we align both exponents r and m on the least significant bit, and divide both exponents in respectively $\lceil \frac{|m|}{k} \rceil$ and $\lceil \frac{|r|}{k} \rceil$ chunks, with $|m|$ and $|r|$ being the length in bits, and compute the following: For each chunk we raise the intermediate result to 2^k and use the chunks from both as an index to Table I, and multiply by that value. In case of the number of bits m being shorter than r, i.e., there are no chunks left for an index, we use 0 for all k bits. By using the pre-computation we save at most $\frac{|m| \cdot |r|}{2} \cdot (k-1)$ multiplication as opposed to standard modular exponentiation, i.e., where it deals only with chunks of $k = 1$. This method for exponentiation is also called 2^k-ary method [29], which we extended to simultaneous multi-exponentiation.

*2) **Mobile Devices - Cloud Manager**:* The encryption manager sends the data to the cloud manager. The cloud manager is responsible for sending the encrypted data from the phone to the cloud data center at the edge of the network (edge cloud). Apart from the REST support, we use a WebSocket connection from the phone to the cloud to deal with real-time sensor data (e.g., accelerometer) with high frequency (100Hz).

*3) **Edge Cloud - Data Processing Manager**:* In the (untrusted) edge cloud a process called the *data processing manager* implements the following functionality. First, it receives encrypted data from the mobile devices and stores it into a query-able store (i.e. ThingSpeak [4]). Second, it receives requests from applications to perform homomorphic operations on (parts of) the data (i.e. data that has been received in the previous N seconds, or the last M received values, etc.). Third, it spawns off a number of *data processing workers* which carry out the requested homomorphic operations. Finally, when the computation has finished, and the result is received from the workers, the manager either stores the result, or sends it to any device that requests it (this could be the device that runs the application that initiated the homomorphic computation, or other devices). Here, we could enforce an authentication mechanism, such that only *accepted* devices can request results, but in practice this may not be necessary because for decrypting results the homomorphic key is needed.

*4) **Edge Cloud - Data Processing Worker**:* Depending on the type of homomorphic computation applied to the data and on the data volume on which the computation is to be performed, the *data processing manager* spawns off a number of *data processing workers*. To decide how many worker nodes are to be spawned we refer the reader to Section V where a detailed evaluation of the system throughput is presented.

The workers carry out the computation in a distributed fashion, splitting the input equally among them. Then, each worker performs the operations on its input data. Depending on the application type, an intermediate reduction phase may be needed to combine the result from each worker. The results are then sent to the manager.

IV. SUPPORTED OPERATIONS & USE-CASES

For our implementation, we use the Paillier [11] cryptosystem. The limitation that this system entails is that it only permits simple arithmetic operations: additions and multiplications. Although this restricts the generality of our proposed solution, there are many sensing applications that can be modeled using these operations. Typically, sensor generated data is represented by numerical (integer) values.

Therefore, applications can compute sums over ranges of values, or averages. This could be achieved over large historical data generated by one or many devices, or in a streaming fashion, using a time-window. Such time-based analyses could

be used for computing trends in user-specific activities and then detect abnormalities for health care. For example, if for the last five minutes, the user heart-rate is elevated, but also the accelerometer sensor indicates significant movement, it could be inferred that the user is training. However, if for the last five minutes the user's heart-rate is unusually high (or low) without any indication of physical activity, it could be inferred that a serious medical emergency is taking place.

An example of smoothing out error is high resolution toxic emissions sensing with IoT devices. To measure the impact of emissions in the environment, many sensors have to be employed to get a high resolution. Such emission measuring sensors have low accuracy (i.e., $\%5$ accuracy). To smooth out errors and outliers in the readings, we can apply a Gaussian kernel over neighboring sensors. Equation 1 shows an example of a Gaussian smoothing kernel. Using Gaussian convolutions we can easily generate more accurate locality- based readings, such as heat maps.

$$\frac{1}{17}\begin{bmatrix}1 & 2 & 1\\ 2 & 5 & 2\\ 1 & 2 & 1\end{bmatrix} \tag{1}$$

Here $a_{k,l}$ is the value of the Gaussian kernel at the k row and the l column. The value $v_{i,j}$ represents the value of sensor value for the coordinate (i, j). Computing a 3×3 Gaussian kernel using a homomorphic cryptosystem is achieved as follows:

$$\begin{aligned}\rho_{i,j} \leftarrow \mathfrak{M} \cdot \left(\sum_{k=-1}^{1}\sum_{l=-1}^{1} v_{i+k,j+l} \cdot a_{k,l}\right) = \\ \mathfrak{M} \cdot D\left(\prod_{k=-1}^{1}\prod_{l=-1}^{1} E(v_{i+k,j+l})^{a_{k,l}}\right)\end{aligned} \tag{2}$$

With:

$$\mathfrak{M} = \frac{1}{\sum_{k=-1}^{1}\sum_{j=-1}^{1} a_{k,j}}$$

We distribute computation over the parties as follows. The encryption of $E(v_{i,j})$ is done by each sensor node. The computation of Gaussian kernel $\sum\sum v_{i+k,j+l}a_{k,l}$ is done by the data processing manager, which to the equivalent to $\prod\prod(E(v_{i,j})^{a_{ij}}$ using homomorphic encryption. Here we assume that the values of the Gaussian kernel $a_{i,j}$ and the kernel it self are public knowledge. The result Gaussian kernel is stored in $\rho_{i,j}$ and can be used to generate image or a heat map.

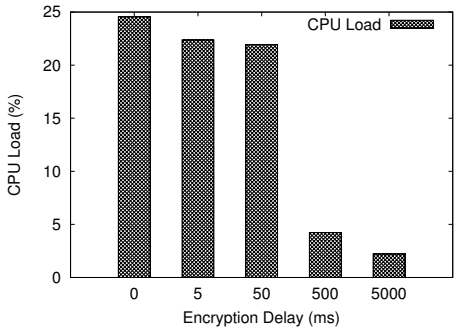
V. Evaluation

In this section we present our experimental evaluation for the P^2-SWAN system. We first show how much overhead is generated by homomorphic encryption on low-power devices. Our evaluation presents power consumption and performance overheads of homomorphic encryption. We also evaluate whether our implementation achieves real-time encryption on such devices.

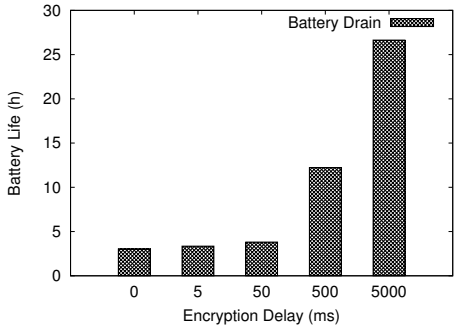
On the edge cloud side, we measure how scalable the platform is when running the applications that perform encrypted operations. We also assess if our proposed solution achieves real-time response times. For the purpose of this paper, we evaluate two applications:

- **Computing Sums/Averages:** The edge cloud receives encrypted values from a million devices. The computing infrastructure performs the homomorphic addition on the encrypted data and sends back the result to the device that can decrypt the result.
- **Gaussian Smoothing Kernel:** The edge cloud receives encrypted values from a million devices. The computing infrastructure performs the homomorphic Gaussian Kernel convolution on the encrypted data and sends back the result to the device that can decrypt the result.

For all applications and benchmarks we use a modulus $|N| = 2048$ bits, and private key of $|\alpha| = 320$ bits. Hence, performance of the decryption is significantly higher than the encryption, since the decryption is in the subgroup variant, i.e., 2048 bits as opposed to the decryption 320 bits.



(a) CPU Load for different encryption delays.



(b) Battery drain for different encryption delays.

Fig. 3. CPU Load and Battery Life for increasingly larger encryption delay on Nexus 5. 0 stands for continuous encryption.

A. Mobile Platform

We evaluate the mobile platform with the following 5 devices:

- LG **Nexus 5** with 2.26 GHz quad-core Qualcomm Krait 400 ARMv7 cortex A9 32bit, 2 GB LPDDR3-1600 RAM running Android version 6.0.1

TABLE II
HOMOMORPHIC ENCRYPTION (2048 BIT) PERFORMANCE ON NEXUS 5 WHEN RUNNING CONTINUOUSLY FOR 5 MINUTES.

#Threads	Encryption Time (ms)	Throughput (encryptions/sec)	Average Power (mW)	Total Energy (mWh)	CPU Load (%)
1	42.71	22.7	2881.89	240.15	24.56
2	55.02	31.14	2998.77	249.89	33.55
3	69.09	37.5	3002.74	250.22	40.42
4	75.11	46.68	4043.63	336.91	53.33

- Raspberry Pi 1 model B (**RPi 1**), 700 MHz BCM 2835 ARMv7 cortex A9 32bit, 256 MB LPDDR2 400MHz RAM running Linux 3.6.11
- Raspberry Pi 2 model B (**RPi 2**), 900 MHZ quad core BCM2836 ARMv7 cortex A7 32bit, 1 GB of LPDDR RAM running Linux 4.4.34.
- Raspberry Pi 3 model B (**RPi 3**), 1.2 GHz quad core BCM2837 ARMv8 cortex A53 64-bit, 1 GB 900 MHz LPDDR2 RAM running Linux 4.4.34 (32-bit mode).
- Scaleway[1] C1 (**SC 1**), 1.333 GHz quad core Armada 370/XP ARMv7 cortex A9 32bit, 2 GB unknown RAM running Linux 3.16.0.

The Nexus 5 is connected to a Wi-Fi router for sending encrypted data to the edge cluster. We keep the device in airplane mode and screen off while running our experiments. We also uninstall or disable other applications running in the phone. The energy consumption and the CPU load is measured using Trepn Profiler [30] tool. The other devices have a wired connection. The devices are currently not considered high-end mobile devices, but rather low to mid range devices.

B. Untrusted Edge Platform

To emulate the untrusted edge computing infrastructure we have used our local DAS-5 [31] cluster[2]. DAS-5 consists of 68 nodes equipped with dual 8-core Intel E5-2630v3 CPUs and 64GB memory. The nodes are interconnected by two networks: an 1Gbps Ethernet, and an 54 Gbps FDR InfiniBand. For all our experiments we have used the latter.

C. Mobile Phone Experiments: Encryption Performance

In this section we present our results for running homomorphic encryption on the low-power devices. On the Nexus 5 phone, we measured not only the encryption performance, but also the energy consumption. On the other devices, we only measured the encryption performance and conducted a **real-time** capability study.

Table II shows the performance measurement on the Nexus 5 device. We run the experiment using 1,2,3 and 4 threads, while the sensor data is continuously sampled for encryption. We note that the encryption time is only **42 milliseconds** when running 1 thread. In comparison, the Polystream [14] (whose source code is not available for implementation comparison) reports **70 milliseconds** for homomorphic encryption on cloud virtual machines. This shows that our implementation achieves better performance even when run on a low-power device.

[1]https://www.scaleway.com
[2]http://www.cs.vu.nl/das5/

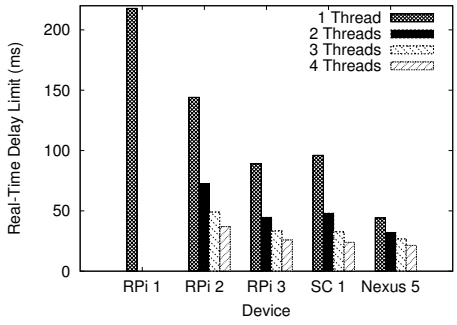
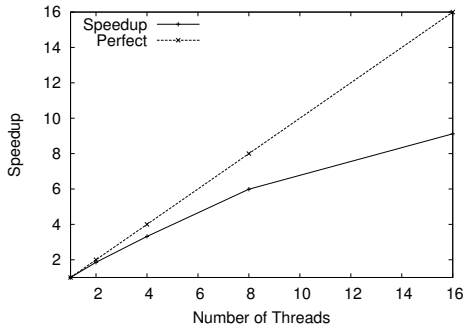
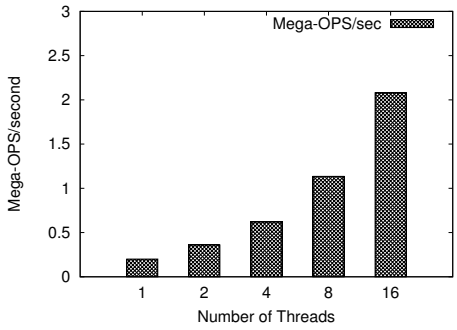


Fig. 4. Real-time delay limit for various low-power devices.



(a) Computing Averages Speedup



(b) Computing Averages Throughput

Fig. 5. Vertical Scalability for Computing Averages over 1 Million Encrypted Entries.

After increasing the number of threads, the encryption time increases due to contention. However, the throughput is more than 2 times higher when running 4 threads compared to 1 thread. The throughput for 1 thread is 22.7 encryption operations in one second. It is important to notice the implication: any sensor data with a frequency lower than 22

Hz can be processed in real time using 1 thread. We show that the normalized (utilization with respect to the maximum CPU frequency) CPU load percentage scales up linearly. We also measure the average power consumption: when running with maximum CPU utilization (4 cores) and maximum data transfer to the edge cloud, the battery drain is approximately 2 hours.

P^2-SWAN in the phone can do local aggregations (e.g., average, maximum, minimum etc.) on the sensor data over a time window. When the time window increases, the frequency of encryption and data transfer to the edge cloud decreases. For example, if a mobile application sends the average sound sensor value over a period 10 seconds, the evaluation engine in P^2-SWAN will keep evaluating the average measurement for a time window of 10 seconds. Then, at each 10 second interval, it sends the result to the encryption manager. The manager encrypts data every 10 seconds and sends it to the edge cloud. Every output data after homomorphic encryption is of the size **421 bytes**. By increasing the time window, we minimize the data transfer cost and the CPU load (for encryption), therefore decreasing the energy consumption. Figure 3 shows the CPU load and the battery drain at various encryption frequencies. We notice that using a time window of 5 seconds leads to 26 hours of battery life and a low CPU load of 2%.

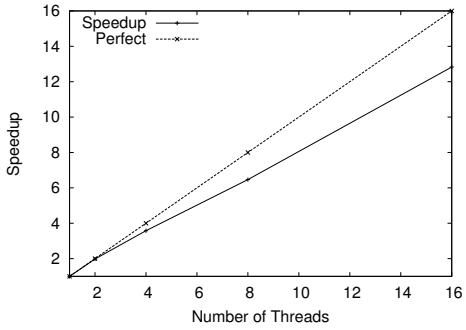
Figure 4 shows our **real-time** evaluation study. We have run our implementation of homomorphic encryption on 5 low-power devices: RPi 1, RPi 2, RPi 3, Scaleway SC 1, and Nexus 5. All devices except the RPi 1 have 4 cores, thus, they can encrypt data in parallel on up to 4 threads. Based on the encryption throughput achieved, we computed the minimum delay at which the devices can poll sensors, encrypt their data and still achieve *real-time encryption*.

We notice that for RPi 1, if the sensor data is sampled less often than 220 ms we can achieve real-time encryption. For the RPi 2 (with 1 thread), the real-time delay is also slightly large: approximately 150 ms. For all other devices (irrespective of the numbers of threads used), the delay for which we can achieve real-time is lower than 100 ms. Furthermore, for all devices, when running with 3-4 threads for performing encryption, real-time is achieved for delays lower than 50 ms.

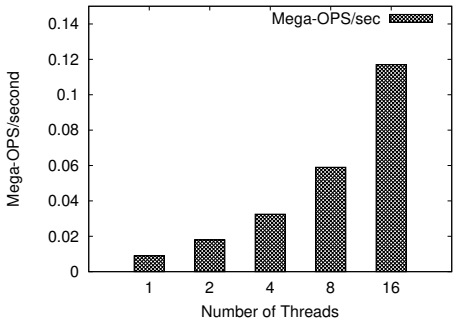
This result ensures that our implementation achieves **real-time encryption** for low-power devices. Usually, IoT devices do not poll sensors continuously. They operate in the following way: (1) poll sensors; (2) send data; (3) sleep for a predetermined amount of time to save battery. Our results show that if the sleeping time is lower than 50-100 ms, **real-time encryption** is achieved. Conversely, even if the sleeping time is smaller than this amount of time, P^2-SWAN helps in achieving real-time by performing local aggregations.

D. Untrusted Edge Platform Experiments

In this section we present our results running the two proposed applications (1) **Computing Sums/Averages**, and (2) **Gaussian Smoothing Kernel** on the untrusted edge platform. Our envisioned setup is straightforward: we assume that we need to run the two applications on **1 million encrypted numbers**. This large number of entries could enter the system in multiple ways: (1) either one million devices send an encrypted value; (2) one device sent a million entries over a period of time; or, (3) a combination of (1) and (2).



(a) Computing Gaussian Kernel Speedup



(b) Computing Gaussian Kernel Throughput

Fig. 6. Vertical Scalability for Computing Gaussian Kernel over 1 Million Encrypted Entries.

Therefore, our evaluation assesses the scalability of our data processing platform when faced with a large-scale load. Also, we want to check whether the computation over 1 million entries can be realized in real-time.

Figure 5 shows the scalability behavior (Figure 5a) and the throughput (Figure 5b) of the **Computing Sums/Averages** application when run only on one data processing worker. As the nodes in our cluster have 16 cores, we run the computation on 1-2-4-8-16 cores. We notice that when gradually increasing the number of worker threads, the speedup is not perfect. The explanation for this behavior is twofold.

First, on one core, the algorithm is running in a sequential fashion, multiplying 4096-bit numbers while also applying a modulo operator on the result. When running on larger numbers of cores, the input is equally split between the workers, but we also need a *reduction* phase to combine the intermediate values computed by each thread. When the number worker threads increases, the reduction phase takes more steps.

Second, our nodes are equipped with two CPUs, each having 8 cores. The two CPUs are organized into two NUMA nodes. When the data gets copied into the node's memory, it gets bound to the memory of only one NUMA node. When running on more than 8 cores, the second NUMA

node performs slower memory accesses through the QPI, thus slowing down the computation. Furthermore, the **Computing Sums/Averages** algorithm is memory-bound: to compute one homomorphic addition, two (4096 bit) loads and one store from/to memory are performed, while there are only two operations applied: a multiplication and a modulo operation. Combined with the cross-QPI memory accesses, this leads to less than perfect scaling behavior

However, the throughput results are encouraging. We notice that when running at full compute capacity (16 cores per node), we reach a throughput of approximately 2 million operations per second. This means that the 1 million entries are computed upon in approximately **0.5 seconds**. Assuming that the data processing manager receives 1 million messages per second from sensors, we believe that this result ensures **real-time** computation with **only one** data processing worker. Therefore, for larger inputs (i.e., given by greater numbers of encrypting devices) it is straightforward to scale the computation on higher numbers of processing workers, and *maintain real-time computation*.

Figure 6 shows the vertical scalability behavior of the **Gaussian Smoothing Kernel** application. In Figure 6a we plot the speedup achieved when gradually increasing the number of worker threads, while in Figure 6b we plot the achieved throughput for the number of Gaussian smoothing operations. First, we notice that the speedup achieved is better than the one for **Computing Sums/Averages** application. The explanation for this behavior is twofold.

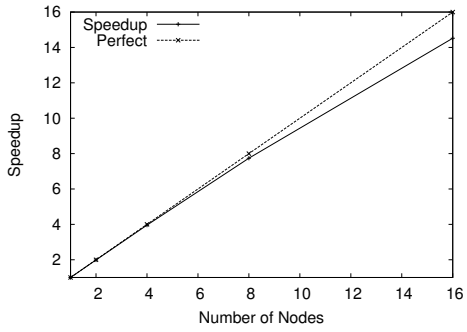
First, it is important to notice that the **Gaussian Smoothing Kernel** application does not need a reduction step when increasing the number of worker threads. This is because the end result is computed for a per-value basis (i.e., for each input value, we have an output value), rather than having just an output value for the entire input. Therefore, there is no need for communication between the worker threads.

Second, the Gaussian kernel is CPU-bound, rather than memory-bound. To compute each output value, we perform $9 + 9$ memory loads and 1 store. However, the number of operations applied to get each output value is larger: we perform 9 exponentiations, 9 multiplications, and for each of these 1 modulo operation. This leads to 36 operations in total, making the computation CPU-bound, as its arithmetic intensity is > 1.

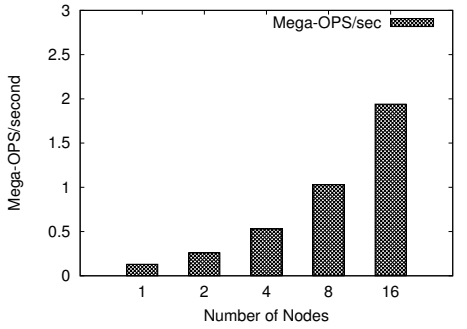
When analyzing the throughput of the Gaussian kernel operations, we notice that we can only compute at most 120.000 *smoothing operations* per second on one node. This is, however, expected, because as we showed earlier, for each Gaussian smoothing operation, we actually apply 36 arithmetic operations. In conclusion, we cannot reach real-time computation for the **Gaussian Smoothing Kernel** application on only one data processing worker.

The solution for achieving better performance is to scale out the algorithm by adding multiple data processing workers. Figure 7 presents our horizontal scaling results: Figure 7a plots the achieved speedup when gradually increasing the number of compute nodes; conversely, Figure 7b presents the achieved throughput. When running these experiments, on each data processing worker we used the configuration that gave the most performance in the single-node setup: each worker runs 16 threads. When performing the experiments on 1-2-4-8-16 workers, we scale out to a total of 256 cores.

When analyzing the results, we notice that the achieved speedup is almost perfect. This is explicable, as the problem is embarrassingly parallel: each node performs the Gaussian smoothing operation on its own partition of the data and there is no need for communication. There is indeed an overlap between workers, as each of them operates on a sub-matrix and when computing the Gaussian values for the top and bottom rows, each node needs values from its adjacent sub-matrices However, this is solved when the data is initially passed to the workers: each gets, at the beginning, its corresponding extra row. The slight degradation in speedup is explained by the extra-communication time needed in the beginning: when broadcasting data to increasing numbers of nodes, more messages need to be passed. This extra added time is, however, negligible, compared to the number of operations performed.



(a) Computing Gaussian Kernel Speedup



(b) Computing Gaussian Kernel Throughput

Fig. 7. Horizontal Scalability for Computing Gaussian Kernel over 1 Million Encrypted Entries.

Figure 7b shows that, with 16 data processing workers we can process approximately **2 million** Gaussian smoothing operations per second. Therefore, processing the 1 million encrypted values input takes approximately **0.5 seconds**. Assuming that the data processing manager receives 1 million messages per second from sensors, using only 16 workers, we achieve **real-time** computation for the **Gaussian Smoothing Kernel** application. If the rate at which the data processing

manager receives messages from sensors increases, *to maintain real-time computation*, the number of data processing workers needs to be scaled proportionally.

VI. CONCLUSION

Typically, while IoT sensors do encrypt their data before transmission, the computational analysis performed on such data is made after decryption. Therefore, IoT ecosystems are prone do data leaks during the data analysis process. To overcome this vulnerability, in this paper we introduced P^2-SWAN, a homomorphic encryption enabled mobile computing framework. P^2-SWAN is composed of two parts: (1) it runs on low-power devices that poll sensors (e.g., mobile phones, ARM-based sensor boards) and (2) untrusted edge (cloud) platforms. On the low-power devices it performs efficient encryption of data readings from sensors, while on the edge platforms it performs encrypted analysis over the sensor-generated data.

Due to our carefully crafted implementation, the impact of encryption on low-power devices is minimized while the achieved performance ensures real-time encryption. At the edge computing platform level, modest numbers of servers are able to handle the load generated by millions of sensors, while achieving real-time data analysis. Therefore, P^2-SWAN is able to ensure secure "blind" computation on sensor-generated data while ensuring **real-time** decision-making.

ACKNOWLEDGMENTS

This work is partially funded by the Dutch public-private research community COMMIT/. The authors would like to thank Kees Verstoep for providing excellent support on the DAS-5 clusters.

REFERENCES

[1] "Cisco Global Cloud Index: Forecast and methodology, 2015-2020 white paper," http://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf, 2015.

[2] V. Cerf and M. Senges, "Taking the internet to the next physical level," *Computer*, vol. 49, no. 2, pp. 80–86, 2016.

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[4] "ThingSpeak," https://thingspeak.com, 2016.

[5] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.

[6] V. Vaikuntanathan, "Computing blindfolded: New developments in fully homomorphic encryption," in *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*. IEEE, 2011, pp. 5–16.

[7] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

[8] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," *SIAM Journal on Computing*, vol. 43, no. 2, pp. 831–871, 2014.

[9] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM, 2011, pp. 113–124.

[10] C. Gentry and S. Halevi, "Implementing gentrys fully-homomorphic encryption scheme," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2011, pp. 129–148.

[11] P. Paillier and D. Pointcheval, "Efficient public-key cryptosystems provably secure against active adversaries," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 1999, pp. 165–179.

[12] T. P. Jakobsen, M. X. Makkes, and J. D. Nielsen, "Efficient implementation of the orlandi protocol," in *International Conference on Applied Cryptography and Network Security*. Springer, 2010, pp. 255–272.

[13] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 85–100.

[14] C. Thoma, A. J. Lee, and A. Labrinidis, "Polystream: Cryptographically enforced access controls for outsourced data stream processing," in *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*. ACM, 2016, pp. 227–238.

[15] N. Palmer, R. Kemp, T. Kielmann, and H. Bal, "Swan-song: A flexible context expression language for smartphones," in *Proceedings of the Third International Workshop on Sensing Applications on Mobile Phones*. ACM, 2012, p. 12.

[16] R. Kemp, "Programming frameworks for distributed smartphone computing," Ph.D. dissertation, Vrije Universiteit Amsterdam, 2014.

[17] R. B. Das, A. van Halteren, and H. Bal, "Swan-fly: A flexible cloud-enabled framework for context-aware applications in smartphones," in *Sensors to Cloud Architectures Workshop (SCAW-2016), held in conjunction with HPCA-22*, 2016.

[18] L. Buttyán and J.-P. Hubaux, "Report on a working session on security in wireless ad hoc networks," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 7, no. 1, pp. 74–94, 2003.

[19] H. Chan, A. Perrig, and D. Song, "Secure hierarchical in-network aggregation in sensor networks," in *Proceedings of the 13th ACM conference on Computer and communications security*. ACM, 2006, pp. 278–287.

[20] H. Suo, J. Wan, C. Zou, and J. Liu, "Security in the internet of things: a review," in *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*, vol. 3. IEEE, 2012, pp. 648–651.

[21] F. Li, B. Luo, and P. Liu, "Secure information aggregation for smart grids using homomorphic encryption," in *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on*. IEEE, 2010, pp. 327–332.

[22] D. T. T. Anh and A. Datta, "Streamforce: outsourcing access control enforcement for stream data to the clouds," in *Proceedings of the 4th ACM conference on Data and application security and privacy*. ACM, 2014, pp. 13–24.

[23] "Kryptographische verfahren: Empfehlungen und schlüssellangen," *Technische Richtlinie TR-02102-1, Bundesamt fur Sicherheit in der Informationstechnik*, 2016.

[24] N. Smart, S. Babbage, D. Catalano, C. Cid, B. d. Weger, O. Dunkelman, and M. Ward, "Ecrypt ii yearly report on algorithms and keysizes (2011-2012)," *European Network of Excellence in Cryptology (ECRYPT II)*, 2012.

[25] A. K. Lenstra and E. R. Verheul, "Selecting cryptographic key sizes," *Journal of cryptology*, vol. 14, no. 4, pp. 255–293, 2001.

[26] N. Saputro and K. Akkaya, "Performance evaluation of smart grid data aggregation via homomorphic encryption," in *2012 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2012, pp. 2945–2950.

[27] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[28] O. Goldreich, *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[29] A. Brauer, "On addition chains," *Bulletin of the American Mathematical Society*, vol. 45, no. 10, pp. 736–739, 1939.

[30] "Trepn power profiler," https://developer.qualcomm.com/software/trepn-power-profiler, accessed: 2016-12-16.

[31] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *IEEE Computer*, vol. 49, no. 5, pp. 54–63, 2016.