

A Key-Switching Protocol for the Homomorphic Encryption of IoT Data

Pratyush Bindal, Behrad Ahmadpour and Himaanshi Bansal

bindalpratyush2010@gmail.com, behrad.ahmadpour@cs.ox.ac.uk, bansalhimaanshi@gmail.com

ABSTRACT

The Internet of Things (IoT) is a growing field, and, as such, we must prioritise the security of data held within these devices due to their vulnerability, as they are commonly low-powered or unprotected. There exist homomorphic encryption schemes that allow secure computations on data from IoT devices. Still, they come at the cost of latency, and those with low latency come at the cost of security. The protocol presented in this paper suggests a way to reduce overhead latency timings whilst maintaining security during data transmission. This is done through key-switching, allowing the data owner to choose parties who are allowed to act on the plaintext rather than the ciphertext.

Additionally, if a party is not trusted, the client and server can continue computing on homomorphically encrypted data, allowing for an uninterrupted workflow. The outlined protocol is created using the OpenFHE Library in C++ and tested on Google Cloud against the protocol in [1]. The protocol improves system performance, with an average 53.86% reduction in average latency and an average 5.98 percentage point reduction in average CPU utilisation.

Keywords: Homomorphic encryption, key-switching, learning-with-errors, OpenFHE, ring-learning-with-errors, lattices, key rings, HELIOS (homomorphic encryption with latency improvements via optimised switching)

INTRODUCTION

Imagine being a scientist working with a highly volatile chemical that evaporates instantly when exposed to open air. To safely study and manipulate it, you place it in a sealed, transparent glovebox. Although you cannot touch the chemical directly, the built-in gloves allow you to interact with it, all while keeping it safely contained. This is analogous to homomorphic encryption, where sensitive data remains encrypted at all times, yet meaningful operations can still be performed on it without ever exposing the underlying data.

This paper presents a protocol built on this principle, making the following contributions:

Box 1: Contributions of the paper

- 1) A protocol that ensures that sensitive data stays encrypted (sealed in the glovebox)
- 2) A protocol that allows meaningful operations (like computations or analysis) to still be performed on the data
- 3) A privacy-preserving framework that ensure that the plaintext data is never exposed whilst simultaneously maintaining its utility

Box 1: Contributions of the paper

Homomorphic encryption and IoT

What is encryption?

Encryption can be defined as a method by which a user can securely share data over an insecure network or storage site [2], preventing unauthorised access. When encrypting data, the aim is to guarantee three main principles:

Box 2: Principles of Encryption

- 1) Protect my (user) information.
- 2) Protect your (client) model.
- 3) Ensure dual-party consent when releasing data.

Box 2: Principles of Encryption

These principles are key concepts that must be met when data encryption is used.

Homomorphic encryption

Homomorphic encryption can be defined as a form of encryption that allows specific types of computation to be performed on ciphertexts and generates an encrypted result which, when decrypted, matches the result of operations performed on plaintexts [3]. In other words: $\text{Dec}(\text{Eval}_f(\text{Enc}(m_1), \text{Enc}(m_2), \dots)) = f(m_1, m_2, \dots)$ [4]. We can use the analogy of an investment portfolio to simplify this definition. When a trader is successful, their brokerage platform will often train its trading models on the trader's portfolio. This data is extremely sensitive and should not be easily accessible. To address this, the trader can encrypt their portfolio using homomorphic encryption before sending it to the brokerage. This allows the platform to train its model directly on the encrypted data, performing calculations without ever seeing the actual contents. The result is a model that benefits from the portfolio data while maintaining the trader's confidentiality.

Additionally, the brokerage may encrypt its model using a second layer of homomorphic encryption, ensuring that when the final model is returned, the trader gains insights or outputs but cannot reverse-engineer the brokerage's proprietary logic. In this way, homomorphic encryption enables mutual protection: the user's data remain private, the platform's algorithms remain secure, and both parties have to agree to the release of the data, as both decryption keys are required for decryption.

A MATHEMATICAL UNDERSTANDING OF HOMOMORPHIC ENCRYPTION

The following summary of the mathematics behind homomorphic encryption includes data from [5].

Homomorphic encryption gets its name from the algebraic term homomorphism, which refers to a structure-preserving map between two algebraic structures of the same type. A scheme is considered somewhat homomorphic (SHE) if it can perform a limited number of operations (either multiplicative or additive) on the ciphertext before losing the ability to decrypt due to noise overlapping with the data buffer. Noise builds up with every homomorphic operation, and after a point, it can overlap the actual ciphertext buffer, resulting in incorrect decryption. A scheme is considered partially homomorphic (PHE) if an unlimited number of computations (i.e., additive or multiplicative) can be performed on the data.

A scheme is considered fully homomorphic (FHE) if it can perform an unlimited number of computations on the data while still maintaining a correct decryption.

There are a few foundational mathematical concepts behind homomorphic encryption:

Algebraic rings

A ring is a mathematical object, like a field, except that not every element in the set has a multiplicative inverse. It is a set with two operations: addition and multiplication. For homomorphic encryption, the commonly used ring is $R_q = \frac{\mathbb{Z}_q[x]}{x^N+1}$, where N is a power of 2. Elements of R_q are polynomials of degree less than N with coefficients in $\mathbb{Z}_q$, where polynomial multiplication is performed modulo $x^N + 1$.

Lattices

Homomorphic encryption schemes are often based upon lattice problems, such as the shortest vector problem (SVP), where one must find the shortest non-zero vector in a lattice, or learning with errors (LWE), where given many noisy linear equations: $a_i \cdot s + e_i = b_i \pmod{q}$ Here, the goal is to recover the secret s , which is hard if e_i is a small noise. This noise contributes to the cryptographic hardness of homomorphic encryption schemes.

Ring - LWE

This is an efficient variant of LWE (Learning with Errors), which works in polynomial rings:

$$a(x) \cdot s(x) + e(x) = b(x) \pmod{q} \text{ where } a(x), s(x), e(x), b(x) \in R_q = \frac{\mathbb{Z}_q[x]}{f(x)}$$

Homomorphic encryption schemes rely on the computational assumption that lattice problems are hard, meaning that recovering the secret $s(x)$ from noisy samples $(a(x), b(x))$ is difficult.

Bootstrapping

To avoid accumulating noise, these schemes use a technique called bootstrapping, where an encrypted secret key and a ciphertext are used to generate an 'equivalent' ciphertext that we can further compute on [6]. Noise occurs as a result of homomorphic operations. Bootstrapping is used to reduce the noise levels created as a result of homomorphic operations, preventing an incorrect decryption. This resets the noise levels of the ciphertext. Steps for Bootstrapping:

1. Encrypt the secret key.
2. Homomorphically evaluate the decryption circuit in the noisy ciphertext.
3. Output a refreshed ciphertext with the same message and reduced noise.

This noise growth is the main mathematical challenge of homomorphic encryption, and a significant reason why it is still not widely used across all industries.

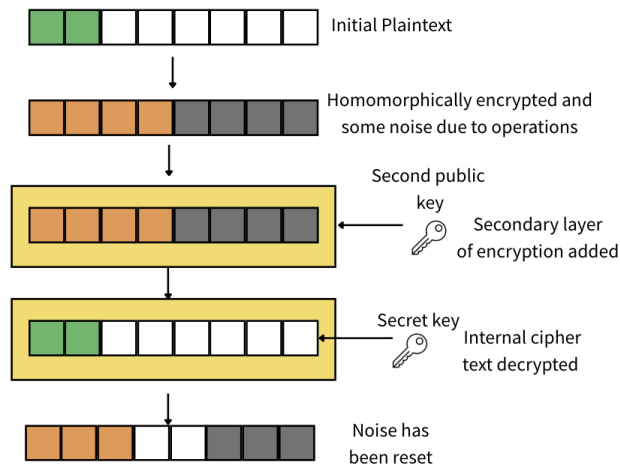


Figure 1. Shows how bootstrapping occurs whilst maintaining data privacy

What is key-switching?

Normally, encrypted data cannot be decrypted without the relevant private key from the same key pair as the public key which was used to encrypt it. Key-switching allows another party to use their own public-private key pair set to switch the encryption to their public key and then decrypt with their private key. This is done by sending a public key from the receiving party to the client. The client then generates what is known as a switching key from the received public key. This switching key is then sent with the homomorphically encrypted data to the receiving party, where they re-encrypt the data using said switching key and then decrypt using their relevant private key, allowing them to access the plaintext of the data. Sending the switching key along with the data is not a concern, as the switching key is tied to the receiving party's keypair, meaning that another party can re-encrypt the data using the switching key but cannot decrypt it, as they do not have the relevant private key. This allows a party to perform mathematical operations on the data faster and more efficiently, as plaintext mathematics is simpler if there is no need for bootstrapping or a modulus with the plaintext modulus. Once the receiving party has worked on the data, they can encrypt it with their own public key and then key-switch using the public key from the original client. This then also permits the client to decrypt the data after they re-encrypt it with the switching key. Key switching is used within fully homomorphic encryption schemes such as OpenFHE alongside bootstrapping to re-linearise data, reduce noise, and perform rotations when required.

Why use key-switching?

As mentioned in the previous section, key-switching has many uses. For one, it can allow for more complex computations on data, as there are no requirements to use homomorphic operations or to worry about reaching a multiplicative depth or overrunning the plaintext modulus. This means that we can have faster, more meaningful and better computations on sensitive data whilst still maintaining encryption. There is also implicit agreement of the three fundamental principles of encryption as seen in box 2, as:

1. The information is protected because it is encrypted and is being sent to a trusted party.
2. The model is protected because the computations performed by it are never seen, and data can be withheld at the discretion of either party, allowing both parties to protect their relevant intellectual property.
3. There is an implicit agreement with the point 3, as both parties consent to release data when they provide the switching key. This action must be intentional and signals consent.

What is IoT?

It is important to understand what IoT is; for this paper, it can be defined as a network of physical devices, vehicles, appliances, and other objects embedded with sensors, software, and network connectivity, allowing them to collect and share data [7]. Furthermore, IoT devices do not have to be connected to a public network to be considered IoT devices; instead, a device within a private network group can still be

considered an IoT device. For example, devices used for monitoring in nuclear power plants [8] or as controllers for automated metro systems [9] are all part of their own IoT. Due to the sensitive nature of this information, a high level of security and discretion is required. Cybersecurity has become a major discussion in the field of IoT, as every new cyberattack (e.g., Mirai Botnet [10]) urges firms to secure their systems further and strengthen their protection.

Cryptography in IoT

To help protect against bad actors using intercepted data, various cryptographic techniques are used [11]. The methods used typically strive to strike a balance between efficiency and security, as many IoT devices lack substantial computational power.

A recent study by IoT Analytics predicts that the number of interconnected IoT devices will exceed 40 billion by 2030 [12]. This growth implies not only an increasing volume of sensitive information on these networks, but also a larger number of potential vulnerabilities for exploitation. As technology continues to evolve, the data protection and encryption standards used everywhere must advance to protect against emerging threats. Homomorphic encryption offers a promising and future-proof approach to data security, capable of withstanding attacks for many years. This paper proposes integrating homomorphic encryption with a secondary cloud-offloading circuit into IoT systems to address these challenges and increase efficiency.

PROBLEM DESCRIPTION

The HELIOS protocol is designed for scenarios in which low-powered IoT devices that produce sensitive data cannot process it alone and therefore offload the processing to other devices over which it may or may not have control.

List of Participants

In this protocol, there are four main types of participants:

1. The client (IoT device)

This is the owner of the data. It generates the data and holds a copy of the first keypair (PK_1, SK_1).

2. Server A (SA)

This is the parent server sitting on the client's side of the network and is responsible for key generation, splitting the data into packets, distributing the packets to its nodes, and managing transmission to Server B (SB). SA is implicitly inside the client's trust domain and can be treated as an extension of the client.

3. Server B (SB)

This is the remote computing party. It receives packets, keyswitches, performs the requested computation and returns results. SB holds its own keypair (PK_2, SK_2) and is trusted only to the extent that the client chooses to trust it.

4. The nodes

These are worker processes attached to SA and to SB, connected over wired links. A node will inherit the trust status of the server that issues the keys it holds, and does not have any independent authority.

A brief comparison against asymmetric encryption

As the protocol relies on key switching, one might argue that the HELIOS protocol is simply a re-enactment of typical asymmetric encryption. However, the HELIOS protocol is unique in the sense that, while it does contain aspects of asymmetric encryption, it allows the trust between parties that allows another party to act on the plaintext data to be revoked at the choice of the data owner. When this happens, the communication process would be halted under asymmetric encryption as the data is rendered useless. However, with HELIOS, homomorphic encryption is used so mathematical computations can continue, and the communication process would continue, albeit at a decreased pace.

To be further explicit, the existence of a switching key indicates the level of consent between the parties. The switching key is distinct and unique to each server's public key and is a clear indicator that this server is allowed to access the data owner's plaintext data. If a switching key is not present between the communication of two parties, then it is clear that either party does not hold a level of trust granting them access to the raw, plaintext data.

Additional assumptions made

In addition to the above, the following assumptions are made when utilising and explaining the HELIOS protocol throughout this paper.

- An attacker cannot read the wired links between a node and its corresponding server. The chance of interception on a local wired connection is treated as negligible, so data on those links does not need to be encrypted.
- The wireless link between SA and SB is public. An attacker can record every message crossing it,
- Secret keys never leave the party that owns them, and the key itself is stored securely at each party.

- The underlying assumption of RLWE problems holds, so ciphertexts reveal nothing about their plaintexts to a party without their corresponding secret key.
- Arguably, the most important assumption is that, under HELIOS, plaintext exposure is not a network-based decision but is rather a decision of the client.

Table of parties

Below is Table 1, outlining the parties, their trust status and their plaintext access under the HELIOS protocol

Party	Trusted?	Sees plaintext?	Can decrypt?	Basis
Client/IoT device	Yes, as it is the data owner	Yes	Yes	It is the data owner and holds SK_1
Server A (SA)	Yes	Yes	Yes	It is inside the client's domain and holds SK_1
Nodes of SA	Yes	Yes	Yes	Inherits the status of SA through their wired link
Server B (SB), assuming authorisation	Conditionally, but in this instance yes	Yes	Yes	Holds SK_2 and, as it is trusted in this scenario, also holds the switching key corresponding to the client's secret key SK_1 and SB's public key PK_2
Server B (SB), assuming unauthorised	No	No	No	Holds SK_2 , but no switching key and therefore computes homomorphically
Nodes of SB	Same as SB	Same as SB	Same as SB	Inherits SB's status through their wired link

Table 1: Parties, trust status, and plaintext access under HELIOS

With this in mind, the question the paper addresses is: can an IoT deployment obtain the latency and CPU behaviour of plaintext computation between different remote systems, without giving up either confidentiality against everyone else or the ability to fall back to fully encrypted communication when authorisation is withheld?

MOTIVATION

Relevancy and importance

In this paper, we discuss increasing the efficiency and safety of IoT devices with offloading or workload-splitting strategies to reduce computational strain on the IoT device in question.

The IoT (Internet of Things) is a rapidly growing, ever-changing field of modern computing, with more interconnected devices integrating into our daily lives. It is quickly becoming a valuable asset in life and affects many aspects of households, businesses, and governments. As a result, the IoT can have a huge impact on our daily lives [13].

Despite advances in the field, homomorphic encryption remains computationally intensive. It is not feasible to perform at a commercial scale on IoT devices without sacrificing latency, efficiency, or accuracy in certain cases. As a result, it has not been widely adopted. Table 2 presents a comparison between traditional encryption methods and homomorphic encryption.

Scheme	Basis	Type	Advantages	Disadvantages
AES	Symmetric block cipher	Standard	Fast, efficient, and widely used	Not homomorphic; data cannot be processed while encrypted
SHA	Cryptographic hashing	Hash	One-way, collision-resistant, ensures integrity	Irreversible; only allows verification against plaintext
RSA	Integer factorization	PHE	Well-known, public-key based, supports signatures	Very slow; only multiplicative homomorphism
BGV	RLWE with modulus switching	FHE	Exact integer operations; supports bootstrapping	Complex implementation; slow for approximate computations
BFV	RLWE-based	FHE	Exact integer arithmetic; suitable for mathematical operations	Noise growth with multiplications; requires parameter tuning
Paillier	Composite residuosity	PHE	Additive homomorphism; simple and efficient	Cannot multiply ciphertexts; large key sizes

Table 2: Comparison of Cryptographic Schemes

Offloading strategies present a viable way to alleviate this pressure on servers and devices by delegating the load onto a holistic set of devices that work in tandem with a parent or controller device/server. However, balancing security, efficiency, and practicality in this situation remains a challenge. The protocol - dubbed HELIOS for "Homomorphic Encryption with Latency Improvements via Optimised Switching" - aims to answer these problems by presenting a situation in which multiparty consent is required for data computation and processing, enabling fluid, efficient, and practical processes whilst not sacrificing security.

Literature review

A fully homomorphic encryption scheme [4] initially proposed bootstrapping to address accumulated noise resulting from homomorphic encryption. This paper created the foundational ideas that allow us even to consider using fully homomorphic encryption in a daily context. This paper, however, does not show a practical representation of bootstrapping in a programme involving homomorphic schemes, but the scheme presented is used in all homomorphic encryption schemes - including HELIOS. Bootstrapping is built into OpenFHE and is performed automatically when required.

Using Fully Homomorphic Encryption in IoT [14]. This paper introduces the idea of using fully homomorphic encryption on IoT devices. It effectively describes the process of encrypting, computing, and decrypting data using fully homomorphic schemes and implements its own variant of it on IoT devices such as an Atmega 2560. Although this paper provides valuable information, its scope remains limited, as it does not implement data communication between IoT constructs, limiting its transition into real-world scenarios where data often need to be processed across distributed systems. HELIOS builds on this foundation by proposing an approach that enables remote computation on encrypted IoT data, ensuring that privacy is preserved even when sensitive data leave the local device.

Benchmarking Homomorphic Encryption on Low-Power Devices: Trade-offs between PHE and FHE [1]. This paper explores the different types of homomorphic encryption. It benchmarks them on low-power devices such as an Arduino to conclude which encryption scheme (either PHE or FHE) gives the best balance between security and efficiency. To benchmark devices, it measures encryption time, decryption time, ciphertext size, memory usage, and energy usage (joules) per 100 operations. The paper cites Microsoft's ElectionGuard [15] as a key situation where the use of homomorphic encryption is warranted, as additive schemes can tally votes without revealing the location or person behind the vote itself. The paper covers PHE schemes such as BFV and Paillier and attempts to test CKKS (FHE) but does not fully attempt to utilise FHE, instead regarding it as unfeasible. Consequently, we reimplement this protocol using the same libraries and systems employed for HELIOS to ensure a fair and controlled comparison. As this protocol is broadly representative of conventional approaches to homomorphic encryption, it constitutes an appropriate baseline and point of comparison for HELIOS. It is also important to note that at the time of writing and editing, this paper is currently in preprint.

Homomorphic Encryption and Network Coding in IoT Architectures: The advantages and future challenges [16]. This paper aimed to determine whether cryptographic techniques such as homomorphic encryption or network coding are feasible to pursue on IoT devices and what challenges need to be solved before we can implement them. The paper effectively presented both techniques along with a thorough comparison of both protocols. It showed many of the problems that we currently face before we can consider a migration to either homomorphic encryption or network coding. The paper did not aim to solve any of these issues, but it aims to address the lack of interoperability with HELIOS through the use of key-switching.

Existing work on homomorphic encryption and IoT offers no practical way to securely offload encrypted IoT data to other parties without sacrificing latency or interoperability. Key-switching through HELIOS closes this gap by letting the data owner selectively authorise trusted parties to compute directly on plaintext, thereby sidestepping bootstrapping and multiplicative-depth limits. The data stays hidden from untrusted parties, and each hand-off requires mutual consent.

The Approach

This paper discusses methods to make homomorphic encryption feasible in IoT/low-powered devices, while facilitating the secure transmission of data between parties whilst maintaining homomorphism. It will also present the overall speed of the process, from encrypting to sending the data, performing the computations, and decrypting locally. This paper aims to demonstrate that fully homomorphic encryption is feasible on low-powered edge devices such as IoT devices with workload-splitting methods, while maintaining a link to reality in terms of latency, efficiency, and security.

Many may suggest that for a protocol such as this, simple asymmetric encryption would be better, as the core principle of local encryption and decryption elsewhere would be achieved. However, there is a fatal flaw with asymmetric encryption, as the ciphertext would become useless without decryption. There may be a scenario in which an untrusted party wants to access some sensitive data from the client and perform computations on it. Homomorphic encryption leaves this possibility open, whereas asymmetric encryption renders this impossible. HELIOS leverages techniques such as key-switching to enable secure computations on plaintext data by ensuring that it is only revealed to trusted parties as decided by the client, protecting sensitive information throughout the entire processing pipeline.

Additionally, if trust is revoked, the plaintext can be swapped to ciphertext, and then the third party can simply compute on the ciphertext rather than the plaintext. In this way, it not only achieves confidentiality but also preserves utility and scalability, advantages that simple asymmetric encryption cannot provide. Furthermore, homomorphic encryption schemes are more resistant to decryption. The NIST talks about the insecure nature of elliptic curve-based encryption schemes due to Shor's algorithm [17], which implies the need for a stronger encryption method to replace these schemes. This paper also states that LWE-based (Learning With Errors) schemes, such as homomorphic encryption, are presumed to be strong because of the hardness of lattice problems, reinforcing the strength of homomorphic encryption over elliptic curve-based schemes such as ECC.

To benchmark the proposed protocol, we have developed it in *C++* and will test various sets of data multiple times. Specifically, we have tested the time taken for a full cycle and also looked at the load on the main server and its nodes. From the HELIOS benchmarks, we have concluded whether homomorphic encryption is possible with our current technological capabilities.

METHODOLOGY

This paper proposes a protocol that allows IoT devices to send data to local servers, which can then handle encryption and data transmission by subdividing the data into multiple packets, each of which is encrypted by an individual node. These nodes also handle homomorphic operations and decryption if required.

Here, there are two servers. The solid lines represent physical wired connections where encryption is not required, as the chance of a bad actor intercepting is very low, and the dashed line represents a wireless connection, where encryption is required. Each server possesses a unique public and secret key. The

server is responsible for key generation and data transmission between nodes, IoT devices, and other servers. The parent server generates private keys for each node and sends them with their public keys, ensuring that each packet of data can only be encrypted to the relevant node. The parent server can then send these data packets to another server, which can then compute on these packets and send them back to the parent server. The secret key of *SB* has been generated by key-switching of the secret key of *SA*, allowing *SB* to decrypt the ciphertext. Each time data are sent from one party to another, key-switching must occur to allow the other party to act on these data. We can observe this in the figure below:

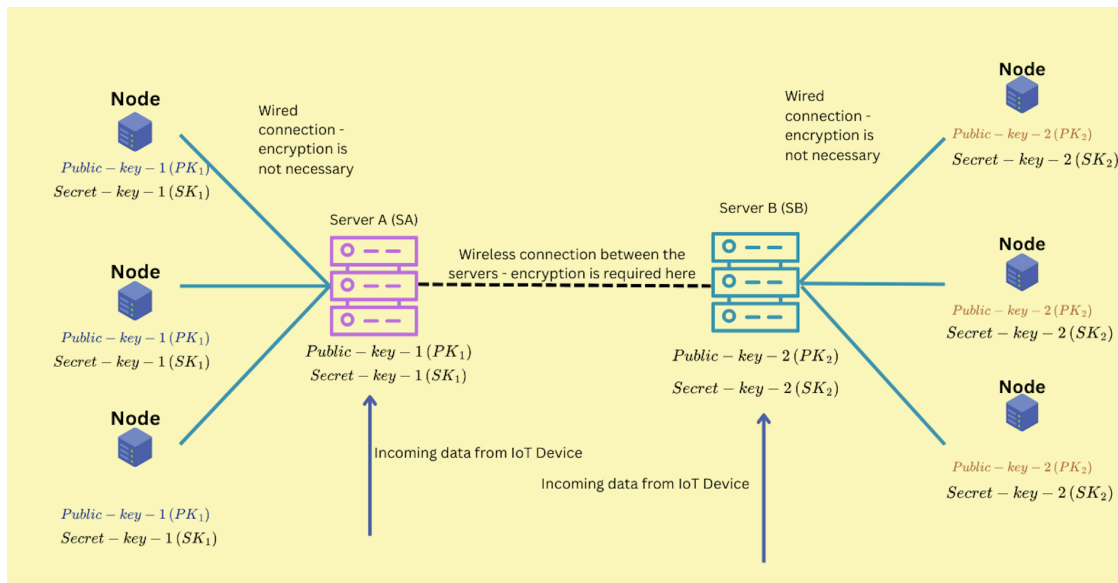


Figure 2. HELIOS protocol

This means that we are meeting the 3 principles as shown in box 2 as:

1. The client information cannot be decrypted unless the client enables the key switch, allowing the other party to decrypt.
2. We cannot interpret the data returned by the other model unless they perform a key switch and allow us to decrypt the data.
3. As both parties have to key-switch, there is an inherent acceptance of point 3 as both parties have consented through the act of the key-switch.

To illustrate the concept of key switching, imagine a medical scenario. A machine records sensitive patient test data, which is encrypted before leaving the hospital. Initially, the data are encrypted with the hospital's secret key, meaning only the hospital can decrypt them.

Using key-switching and switching keys, encrypted test results can be handed off between parties, each

able to decrypt with their own secret key or compute without decryption if the receiving party does not hold that level of trust. The hospital controls who can ultimately interpret the sensitive information by deciding which laboratories generate the switching keys.

Method of benchmarking

To benchmark the protocol, it was programmed in C++ and simulated SA and SB using the Google Cloud Platform. To simulate SA and SB, we also measured the time taken for a full cycle (key generation, encryption, key switch, data transmission, operations, data transmission, and decryption) and used the Google monitoring agent to record system usage. Each data set was tested multiple times; an average was taken to account for anomalous results. An important distinction must be made. The key pairs for each server are made before the timer starts, and any switching keys generated during the data processing and computation are included in the timer.

To actually perform homomorphic encryption, this paper uses the OpenFHE library [18] as it is a fully homomorphic encryption scheme and also has a unique characteristic known as key-switching, which has allowed us to decrypt a homomorphically encrypted piece of data with a second secret key SK_1 with a switching key, allowing two parties to decrypt the same piece of data.

For the programme itself, this paper uses the homomorphic encryption standard *BGV* based on the benchmarking performed in [1], as their paper presents it as efficient, fast, and secure compared to other algorithms tested, such as *CKKS* or *BFV*. A plaintext modulus of 131071 is used, with a multiplicative depth of 3 and a ring dimension of 16384. In addition, the version of OpenFHE used to conduct the experimentation was 1.3.1.

This paper compares the HELIOS protocol with the protocol presented in [1]. However, instead, we programmed the pseudocode of their protocol using the same framework we used for HELIOS and conducted the same test with their protocol and HELIOS, allowing us to test the quality, efficiency, and speed of both protocols while reducing factors that may give another party an advantage. Each protocol was run 50 times with the same set of data, and the average time was then taken. The five random four-digit positive and negative numbers are read from a file. For each set of 50 results, the average CPU (%), Memory (%), and Time (ms) are taken. After this, another set of 50 readings is taken. This process repeats until there are 3 sets of concordant results, with each set averaged to give a final result that can be used for comparison and conclusion. By taking a large number of readings and averaging them, the impact of random error on the results is greatly reduced, increasing the final accuracy of the results and, by extension, the conclusions drawn from this data.

To reduce inconsistent times caused by reading the file, the timer will start once the client or IoT device receives the data. Additionally, the wireless aspects of both protocols have been removed, as this adds another variable which can result in skewed or incorrect results. The internet is not something this paper can control, as there can be random disconnections or other network changes and instabilities that may

result in unfair advantages or disadvantages towards a certain party. Both servers are identical in their specification, so any differences in results will be a result of the individual protocol and not the server itself. These specifications were chosen because a company or provider handling sensitive IoT devices would typically have access to much larger, more expensive, and faster computing units, so this was done to be conservative while still highlighting the differences between the protocols.

With this controlled experimental design, the protocol was subjected to calibrated and repeated testing, resulting in a robust validation of the results presented herein and supporting their reproducibility and independent verification.

Timing procedure

End-to-end latency was measured on the host running the protocol using the C++ “high_resolution_clock.” The timer starts once the client has received a data vector from the input queue so that file reading does not contribute to the measurement. The timer then stops once the final plaintext value has been reconstructed at the client following the return key-switch. The measured window therefore covers encryption of each value, generation of switching keys, transfer to the server, re-encryption and decryption at the server, the arithmetic test, re-encryption and key-switch back, and decryption at the client. Keypair generation falls outside this window. Each of the 50 rows in the input data file is timed individually, and the mean of those 50 timings is then reported as one result. CPU and memory utilisation were sampled independently of this timer, at five-second intervals, using a monitoring script running alongside the protocols.

Handling Multiple Requests

When a server that runs a model like this has an API, it is common for multiple simultaneous requests to be made for the model to act on their data. To avoid overloading issues, each API request is added to a queue that performs the next request as soon as the previous request completes execution. This is how the programme works: the file is loaded into a queue of vectors, with each vector containing five numbers.

The Mathematical Test

To test the protocols on the Google Cloud server, we added a load to the system that involves a computation on the numbers given to the respective protocols. Each run of the full protocol uses five generated numbers as input, so we had to choose something feasible in BGV with a plain-text modulus of 131071. The following test was used:

Input: Ciphertexts $\{c_0, c_1, \dots, c_{n-1}\}$

Output: Computed data $ctTotal$

```

 $ctTotal \leftarrow c_0$  for  $j \leftarrow 0$  to 3 do
  for  $i \leftarrow 1$  to  $c.size()$  do
    for  $r \leftarrow 0$  to 20 do
      |  $ctTotal \leftarrow c_i$ 
    end
  end
  for  $i \leftarrow 1$  to  $c.size()$  do
    for  $r \leftarrow 0$  to 20 do
      |  $ctTotal \leftarrow -c_i$ 
    end
  end
end
for  $i \leftarrow 1$  to  $c.size()$  do
  for  $r \leftarrow 0$  to 20 do
    |  $ctTotal \leftarrow c_i$ 
  end
end
for  $i \leftarrow 1$  to  $c.size()$  do
  for  $j \leftarrow 0$  to 20 do
    |  $ctTotal \leftarrow -c_i$ 
  end
end
end

```

In essence, the programme simulates a large number of addition and subtraction cycles to simulate the effect of a large model acting on homomorphically encrypted data. The code sample above shows the code for the other protocol as shown in [1]. For the HELIOS protocol, the code test is equivalent and acts on the plaintext rather than the ciphertext. To make it as close as possible, after each mathematical operation, the modulus of the result by the *plaintext modulus* is taken, as that is what happens after every operation in a homomorphic system.

For example: $total = (total + decData[i]) \bmod modulus$, where *modulus* is 131071.

This has allowed us to stress the speed of HELIOS and compare it with the other protocol proposed in [1]. Its simplicity is deceptive, as the repeated addition and subtraction of five numbers may seem trivial when in the context of raw data on a modern system. Still, it can be extremely taxing on a homomorphically encrypted system. If something as simple as addition is faster when using the proposed protocol, then larger, more intensive workloads may also be streamlined and more efficient, which this paper aims to simulate but cannot fully replicate.

Examples of key-switching

Key-switching has already been adopted in various aspects of homomorphic encryption. For example, schemes such as CKKS [19] and BFV employ key-switching during their bootstrapping procedures to transform bootstrapped data back into usable data using the evaluation key. HELIOS differs from this

automatic approach by using key-switching explicitly to ensure multiparty communication (MPC), thereby reducing common issues with homomorphic encryption, such as interoperability and latency.

Keyrings

A keyring is a file or database that stores multiple cryptographic keys, including public keys, private keys, digital certificates, and trusted root keys. Keyrings have many purposes, such as managing digital identities and securing communications. To understand the concept of a keyring in cryptography, we can use the analogy of a physical keyring. A physical keyring holds multiple different keys, each of which serves the purpose of opening a unique lock. A digital keyring holds various keys for different encryption or authentication purposes, along with the organisation and easy access of these cryptographic secrets. Keyrings also have many developer benefits. When a keyring is created, it can be exported or imported between environments - assuming security is in place - to allow for simplified deployments or transitions between systems.

Keyrings in homomorphic encryption

When performing homomorphic encryption, it often requires a large number of unique keys. For example, the public key, private key, re-linearisation key and key-switching key. A keyring would be very useful in this scenario, allowing a developer to organise these keys and have easy access to them. It makes management easier, as a programmer can tell a system to simply load a keyring rather than load every single key individually. Additionally, this makes it easier to perform homomorphic encryption at scale, where there are hundreds of keys relating to different clients, servers, devices, or pieces of data. Using a single structured keyring dramatically reduces the chance of mismatching or missing keys. Most homomorphic encryption schemes, such as OpenFHE, use key rings internally but label them as context or a set of keys. In OpenFHE, we defined a key pair for both protocols, which was then added to a key ring upon which keys were added, such as re-linearisation keys:

```
Input : CryptContext cc  
Output : Key-pair keyPair  
// Public + Secret key  
keyPair  $\leftarrow$  cc  $\rightarrow$  KeyGen()  
// Relinearisation keys  
cc  $\rightarrow$  EvalMultKeyGen(keyPair.secretKey)  
// Key-switching keys  
cc  $\rightarrow$  EvalAutomorphismKeyGen(keyPair.secretKey)
```

Pseudocode

The HELIOS protocol

In the following, one can observe the pseudocode for HELIOS, with the raw code available at [20]:

```

Input: Plaintexts  $\{p_0, p_1, \dots, p_{n-1}\}$ 
Output: Computed data  $ctTotal$ 

Client:
  Distribute  $\{p_i\}$  between nodes
  foreach Node do
    Encrypt  $p$  into  $c$  using public key  $pk_1$ 
    Fetch server public key  $pk_2$ 
    Generate switching key  $swk_1$ 
  end
  Send data to server

Server:
  Define plaintext vector  $vk_1$ 
  Delegate each ciphertext with its switching key to a node
  foreach Node do
    Re-encrypt  $c$  with switching key  $swk_1$ 
    Decrypt  $c$  into  $p$  using secret key  $sk_2$ 
    Add  $p$  to plaintext vector  $vk_1$ 
  end
  Perform computations on ciphertexts
  Distribute computed data among nodes
  foreach Node do
    Encrypt  $p'$  into  $c'$  using public key  $pk_2$ 
    Fetch client public key  $pk_1$ 
    Generate switching key  $swk_2$ 
  end
  Send processed data and switching keys to client

Client: Define plaintext vector  $vk_2$ 
  Distribute  $\{c'\}$  between nodes
  foreach Node do
    Re-encrypt  $c'$  with switching key  $swk_2$ 
    Decrypt  $c'$  into  $p$  using secret key  $sk_1$ 
    Add  $p$  to plaintext vector  $vk_2$ 
  end
  Aggregate results into final output  $R$  return  $R$ 

```

Other protocol

Below is the pseudocode for the other protocol as seen in [1]. In this paper, only a graphic is provided, so we have transcribed it into pseudocode. The raw code variant is available at [21]

```

Input: Plaintext message
Output: Final decrypted message after processing
Select Ring Dimension  $n$  Select Ciphertext Modulus  $q$  Select Plaintext Modulus  $t$  Select Fresh Modulus  $p$  Select
Noise Distribution  $\chi$  Select multiplicative depth;
if key-pair does not exist then
    | Generate keys
end
// This all happens automatically with BGV in OpenFHE
Get data from IoT device if noise level exceeds threshold then
    | Perform modulus switching Perform re-linearisation
    | if still required then
    | | Perform bootstrapping
    | end
end
Use secret key  $sk$  to decrypt  $c$  Compute decryption  $M' = Dec(sk, c)$  Verify noise is within bounds
return  $M'$ 

```

EXPERIMENTATION

Below are the results from the protocols being conducted on Google Cloud servers. Each server is configured with the same hardware as in the Google Cloud console, with eight AMD Genoa vCPUs, 32GB of memory, and 50GB of SSD storage.

Protocol	Repeat no.	CPU (%)	Memory (%)	Time (ms)
HELIOS Protocol	1	50.43	7.29	410.06
HELIOS Protocol	2	50.51	7.29	410.50
HELIOS Protocol	3	50.86	7.34	409.52
HELIOS Protocol	Average	50.60	7.31	410.03
Other Protocol	1	56.38	2.20	887.56
Other Protocol	2	57.08	2.13	887.30
Other Protocol	3	56.28	2.11	890.92
Other Protocol	Average	56.58	2.15	888.59

Table 3: Benchmark comparison of protocols

Comparison of data

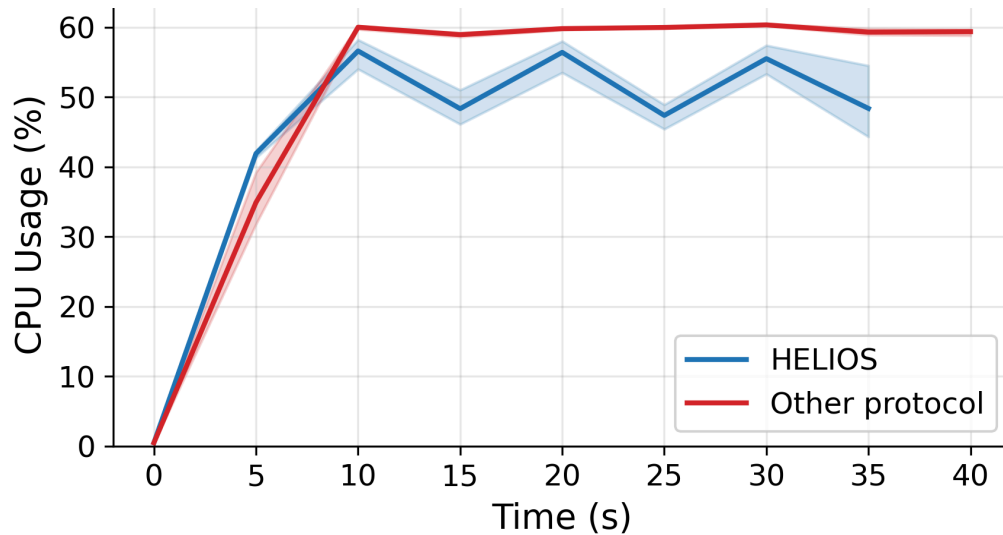


Figure 3. Comparison of CPU Usage between the protocols

Figure 3 represents a comparison between the average CPU usage of both protocols in the three repeated tests performed for all protocols over time. The shaded band around the solid line shows the range of results at that particular time, with the solid line representing the average value. It may be noticed that the graph for the protocol ends abruptly before that of the other protocol, as HELIOS finished faster and did not have data for the next five seconds. When looking at the data, we can observe that overall there is a maximum of a 12.6 percentage point difference in CPU utilisation; at a minimum there is a 3.4 percentage point difference. Additionally, the load on the server controlling the other protocol is much more consistent and level, whereas the CPU utilisation of the HELIOS protocol alternates in an almost sinusoidal fashion. Regardless of this variance, the presented protocol clearly has lower CPU usage compared to the other protocol. When we consider the overall average CPU usage of both protocols as a single figure, as seen in Table 3, the HELIOS protocol has a 5.98 percentage point advantage in terms of CPU resource consumption.

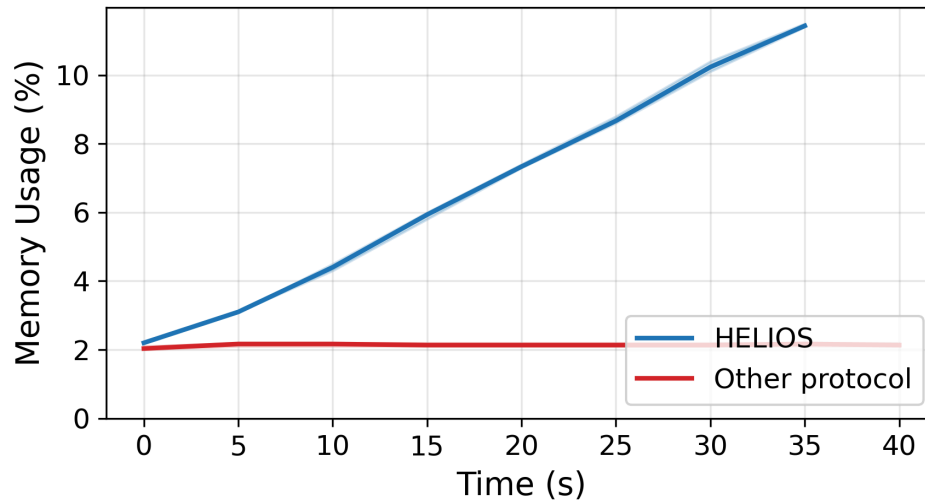


Figure 4. Comparison of memory usage between the protocols

Figure 4 represents a comparison between the average memory usage of both protocols over time. The shaded band, although barely visible in the memory trials, shows the range of results at that time, with the solid line representing the average value. Again, it can be noticed that the line representing HELIOS ends before the line of the other protocol, since HELIOS finished faster than the other scheme. However, the other protocol has far lower RAM usage, drifting from an idle of 2.03% by only about 0.14 percentage points to 2.17% at maximum. On the other hand, the HELIOS protocol reached a maximum RAM utilisation of 11.43% as seen above in Figure 4, corresponding to a maximum difference of 9.40 percentage points between the two protocols. A variety of factors could cause this stark difference in RAM utilisation. Still, the main reason may be that key-switching produces multiple distinct data vectors, rather than a single packed plaintext as in the other protocol. This means more data are being stored in RAM, and memory may not be released before the next set of data runs. The other protocol, as mentioned above, simply took the vector and encrypted it as one plaintext. In contrast, HELIOS broke up the vector into a vector of ciphertexts and a vector of switching keys, the latter taking up a significant space of memory.

Taken together, the results above present a clear latency-versus-memory trade-off rather than framing an outright superior methodology of performing homomorphic encryption. HELIOS is the better choice when end-to-end latency or CPU load is the binding constraint and the controlling server has RAM to spare. In addition, HELIOS completes a full encryption cycle in roughly half the time (410.03 ms versus 888.59 ms), making it well-suited to time-sensitive or interactive workloads or in multiparty settings where data must be securely handled between parties, trusted and untrusted alike. The baseline protocol from [1] remains preferable when memory is scarce, as it has lower RAM utilisation than HELIOS.

In short, HELIOS trades higher memory consumption for substantially lower latency and CPU usage, so

the decision between which protocol to use reduces to which resource a given deployment can least afford to spend. An environment of high-RAM, low-CPU that requires high-speed yet secure computation would benefit from HELIOS compared to the protocol in [1]

The raw data, charts, graphs and figures for the HELIOS protocol are available at [22]

An important note

One might argue against the validity of this paper, as it does not provide a fair comparison with the baseline protocol presented in [1]. This is because the baseline protocol, as simulated using the code from [21], performs computations on homomorphically encrypted data. In contrast, the HELIOS protocol, as simulated using the code from [20], uses key switching and, as a result, performs computations on plaintext data.

This is a fair argument; however, it misrepresents the protocol and its aims. The HELIOS protocol allows plaintext computations only when both parties agree, as shown in Box 2. When trust is revoked, the protocol falls back to ciphertext computations, resulting in a processing flow nearly identical to that of the baseline. As a result, this paper presents data from the HELIOS protocol under the assumption that the party performing the computations is trusted and therefore permitted to compute on plaintext data. When trust is revoked, the protocol reduces to conventional homomorphic evaluation, and its performance is therefore expected to be comparable to the baseline.

The HELIOS protocol is unique from the perspective of secure data holders, who must keep their data flowing through various systems, even when parties cannot be trusted. Unlike traditional asymmetric encryption, or the baseline protocol in [1], HELIOS supports both scenarios without the need to switch encryption standards or halt data processing entirely. This is what makes HELIOS a viable alternative to conventional homomorphic encryption.

The table below outlines what each of the computing parties can see under each protocol.

	HELIOS (Authorised)	HELIOS (Unauthorised)	Baseline protocol [1]
Form of the data during computation	Plaintext	Ciphertext	Ciphertext
Learns the input value?	Yes	No	No
Learns the computed results?	Yes	No	No
Requires an act of consent from the data owner?	Yes, in the form of a switching key	Not applicable	Not applicable

Data remains confidential during transit?	Yes	Yes	Yes
---	-----	-----	-----

Table 4. What the computing party observes under each protocol

Security tradeoff

The protocols also differ in what they require the data owner to give up, as highlighted in Table 4. Under the baseline, the computing party holds only the ciphertext; it learns neither the inputs it computes on nor the result it produces. Under HELIOS, a computing party that has been issued a switching key decrypts before it computes, and therefore sees both. The latency and CPU figures reported above in Figures 3 and 4 are evaluated under those conditions, so they measure the saving available to an owner who is willing to grant it, rather than the efficiency of computing under encryption. What HELIOS retains is confidentiality against every party the owner has not authorised, and a mechanism that makes the authorisation explicit, bound to one recipient, and issued per exchange rather than at once. What it does not retain is the computing party's blindness to the data. The baseline therefore remains preferable wherever the computing party cannot be trusted with plaintext at all, whatever the latency cost, and HELIOS is appropriate only when the decision to trust has been made.

CONCLUSION

In the benchmarking performed in Table 3, HELIOS delivers a substantial performance improvement over the baseline protocol from [1], cutting average end-to-end latency by 53.86% (410.03 ms versus 888.59 ms) and reducing average CPU utilisation by 5.98 percentage points across repeated trials.

This gain comes at a cost: the HELIOS protocol decomposes each input into separate vectors of ciphertext and switching keys rather than a single plaintext pack, resulting in a considerable increase in memory consumption, with HELIOS drawing on average 5.16 percentage points more memory than the baseline protocol. In practice, for IoT deployments, this reframes the protocol choice as a trade-off between latency and memory rather than as a universally superior choice. HELIOS is the stronger option for time-sensitive, interactive workloads and for multiparty settings where data must be securely handed between trusted and untrusted parties, whereas the baseline remains preferable on tightly memory-constrained hardware

Therefore, HELIOS in this paper suggests a viable alternative to conventional homomorphic encryption for deployments willing to authorise a computing party, preserving confidentiality against every party the data owner has not authorised whilst maintaining the ability to compute on the data regardless.

Potential future research

This paper does not address the multitude of real-world costs of homomorphic encryption, such as power, labour for setup, or maintenance costs. A future paper could create guidelines for homomorphic encryption, allowing industries to select the scheme that gives them the best blend between speed,

security, and cost. Future work could also test the HELIOS protocol on real IoT Hardware (such as a Raspberry Pi) or evaluate the practicality of trust revocation mid-computation, a potential use case outlined but not tested in this paper.

REFERENCES

- [1] Het Khatusuriya, Dhvani Patel, Martin Parmar et al. Benchmarking Homomorphic Encryption on Low-Power Devices: Trade-offs Between PHE and FHE, 01 July 2025, PREPRINT (Version 1) available at Research Square [<https://doi.org/10.21203/rs.3.rs-6964665/v1>]
- [2] D. Boneh, A. Sahai, and B. Waters, “Functional encryption: Definitions and challenges,” in *Theory of Cryptography Conference*, Springer, 2011, pp. 253–273. doi: https://doi.org/10.1007/978-3-642-19571-6_16.
- [3] X. Yi, R. Paulet, and E. Bertino, “Homomorphic encryption,” in *Homomorphic encryption and applications*, Cham: Springer International Publishing, 2014, pp. 27–46.
- [4] C. Gentry, A fully homomorphic encryption scheme. Stanford University, 2009.
- [5] R. Meissen, “A mathematical approach to fully homomorphic encryption,” Major Qualifying Project Report, Worcester Polytechnic Institute, 2012.
- [6] A. Al Badawi and Y. Polyakov, “Demystifying bootstrapping in fully homomorphic encryption,” *Cryptology eprint archive*, 2023.
- [7] IBM, “What is the internet of things (IoT)? - IBM.” ["https://www.ibm.com/think/topics/internet-of-things"](https://www.ibm.com/think/topics/internet-of-things).
- [8] A. Ali, Y. Gudarzi, B. Abouabdellah, T. Elzayat, M. F. Ali, and G. Saini, “Smart nuclear power plants operating system through IoTs,” *International Journal of Engineering Research & Technology*, vol. 11, no. 4, 2022.
- [9] P. Fraga-Lamas, T. M. Fernández-Caramés, and L. Castedo, “Towards the internet of smart trains: A review on industrial IoT-connected railways,” *Sensors*, vol. 17, no. 6, 2017, Available: <https://www.mdpi.com/1424-8220/17/6/1457>
- [10] “What is the Mirai botnet? | Cloudflare.” ["https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/"](https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/).

- [11] Ms S. R., “Cryptanalysis for various cryptographic security algorithms using big data in cloud security,” in International conference on contemporary researches in engineering, science, management amp; arts, 2020, Bonfring, Feb. 2020, p. 13. doi: [10.9756/bp2020.1002/13](https://doi.org/10.9756/bp2020.1002/13).
- [12] S. Sinha, “Number of connected IoT devices growing 13% to 18.8 billion.” IoT Analytics; <https://iot-analytics.com/number-connected-iot-devices/>.
- [13] J. Dheeba, V. Oberoi, R. R. Singh, and V. G. Karthik, “Securing electrical drive systems against man-in-the-middle attacks using s-box optimized AES encryption,” IEEE Access, vol. 13, pp. 114716–114735, 2025, doi: [10.1109/ACCESS.2025.3584926](https://doi.org/10.1109/ACCESS.2025.3584926).
- [14] Z. E. Temirbekova, G. Turken, and M. M. Barata, “Using fully homomorphic encryption in IoT.” in DTESI (workshops, short papers), 2023.
- [15] “What is ElectionGuard? - ElectionGuard.” <https://www.electionguard.vote/>.
- [16] G. Peralta, R. G. Cid-Fuentes, J. Bilbao, and P. M. Crespo, “Homomorphic encryption and network coding in IoT architectures: Advantages and future challenges,” Electronics, vol. 8, no. 8, p. 827, 2019.
- [17] National Institute of Standards and Technology, “Post-quantum cryptography standardization project.” <https://csrc.nist.gov/projects/post-quantum-cryptography>.
- [18] A. A. Badawi et al., “OpenFHE: Open-source fully homomorphic encryption library.” Cryptology ePrint Archive, Paper 2022/915, 2022. Available: <https://eprint.iacr.org/2022/915>
- [19] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in Advances in cryptology – ASIACRYPT 2017, Springer, 2017, pp. 409–437. doi: <https://doi.org/10.1007/978-3-319-70694-8>.
- [20] P. Bindal, HELIOS Protocol. (Sep. 2025). Available: <https://github.com/PatB1234/ResearchPaperProtocol>
- [21] P. Bindal, Protocol - Khatasuriya et al. (Sep. 2025). Available: <https://github.com/PatB1234/OtherProtocol>
- [22] P. B. et al., “HELIOS protocol data set.” IEEE Dataport, 2025. doi: [10.21227/wn4c-zr59](https://doi.org/10.21227/wn4c-zr59).

Appendix A: Data from both protocols

Below is a collection of charts displaying the data from both protocols.

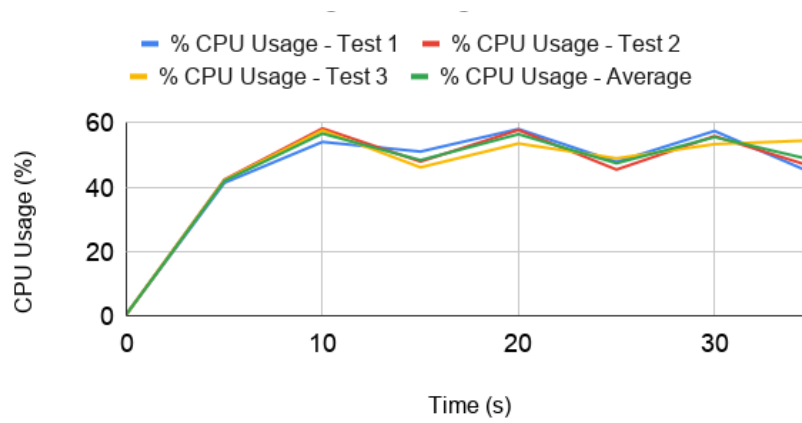


Figure A1. CPU Usage of HELIOS

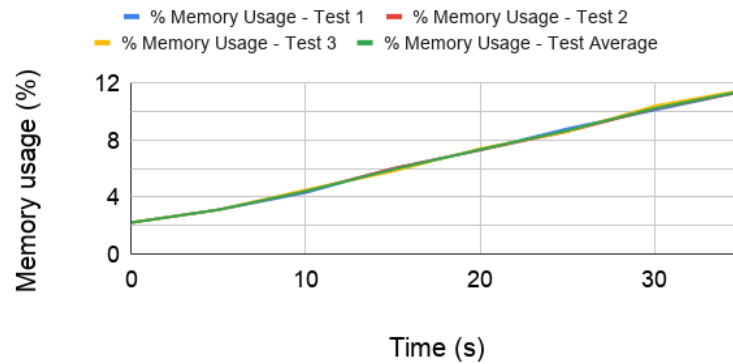


Figure A2. Memory usage of HELIOS

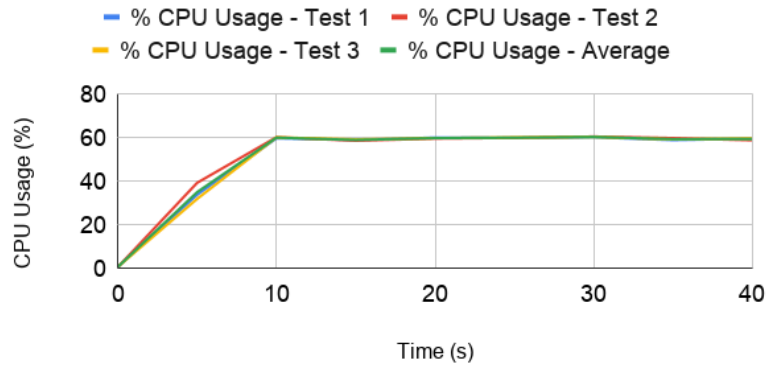


Figure A3. CPU Usage of [1]

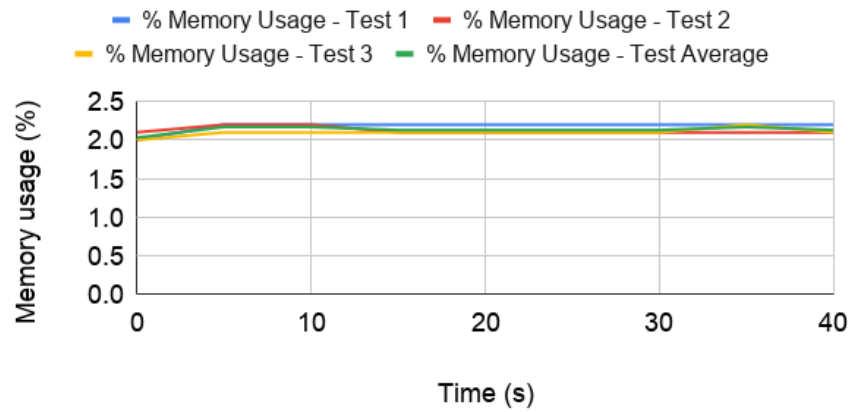


Figure A4. Memory usage of [1]