ITU Journal
Future and evolving technologies
ITU Journal on Future and Evolving Technologies, Volume 6, Issue 3, September 2025
ARTICLE

Challenges with handling keys for secure AI

***Akram Bitar*[1], *Greg Boland*[2], *Nir Drucker*[1]**

[1] *IBM Research - Israel*, [2] *IBM Research - Yorktown Heights, NY, USA*

Corresponding author: Akram Bitar, akram@il.ibm.com

Secure generative AI using Fully Homomorphic Encryption (FHE) enables the utilization of cutting-edge AI capabilities while preserving the privacy of both the model and user queries. Despite its promise, the development and maintenance of scalable solutions in this area remain obscure with challenges that require further exploration. In this paper, we examine key management challenges that are often overlooked in modern research, particularly in offline flows. We identify gaps in current NIST KMS standardization, introduce the concept of a Hierarchical Key Management System (HKMS) solution, and share insights from a recent demonstration of IBM's He4Cloud design.

Keywords: Generative AI, homomorphic encryption, infrastructure setup, key management

1. INTRODUCTION

Generative AI enables creating content based on input from a user: text, images, video, audio, or software code, where the user's input can include private sensitive information. This data is most likely something that should not be freely moving around the Internet unprotected. Clearly, one possibility is to not move data anywhere and leave it where it is. However, while this solution solves the issue of protecting the data, it may block the user from using the newest AI capabilities. Particularly, a main issue preventing enterprises from implementing this solution is having to develop a system of machine learning in their own infrastructure. Developing their own models is time intensive and expensive, not to mention requires an amount of data that they might not have access to easily or legally. It is difficult for a business that is not large enough to support these costs to train on their own. However, if they rely on others that have done the work training models and are willing to lease or lend their models for use, then this becomes a reality.

Notably, enterprises who have trained models often consider this data as valuable Intellectual Property (IP) that they want to protect as well. That helps explain why they are hesitant to let it leave their servers. This can be seen on any attempt to interact with generative AI. ChatGPT[1], Claude[2], Copilot[3], and most other generative AI models do not leave their original servers.

This can change if a model or a query is encrypted before use, and it is exactly what can happen if FHE is used. Informally, an FHE scheme is a cryptosystem that allows its users to evaluate any circuit (function) on encrypted data (see a survey in [1]) using the following four methods: $Gen, Enc, Dec, Eval$. The client uses Gen to generate a secret key sk together with an associated public key pk and an evaluation key ek. Using pk and ek, all parties can encrypt sensitive data m_i by calling $c_i = Enc_{pk}(m_i)$. Subsequently, the user can ask an untrusted entity to execute the function $c_{res} = Eval_{pk,ek}(f, (c_1, \ldots, c_n))$ in order to evaluate a function f on some ciphertexts c_i and store the results in another ciphertext c_{res}. To decrypt c_{res} using sk, the user calls $m_{res} = Dec_{sk}(c_{res})$. An FHE scheme (e.g., BGV [2], B/FV [3, 4]) is correct when

1 https://chatgpt.com/
2 https://claude.ai/
3 https://copilot.microsoft.com/

+

$$f(m_1, \ldots, m_n) = Dec(Eval(f, Enc(m_1), \ldots, Enc(m_n))) \quad (1)$$

and is approximately correct (e.g., CKKS [5]) when

$$f(m_1, \ldots, m_n) = Dec(Eval(f, Enc(m_1), \ldots, Enc(m_n))) + \epsilon \quad (2)$$

for some small ϵ.

Scenarios. Typically, there are two scenarios, where we defer key-related discussion to a later section: 1) encrypting a model; 2) encrypting the query. In the first scenario, the model owner homomorphically encrypts their model and then sends it to another server. The other server, which is called the leasing server, can construct the environment to their own specifications, and input their data to the encrypted model if they wish to run an inference operation. This lends results that are also encrypted. In order for the leasing server to see their results they need to decrypt using the secret key. But if the model owner gives over the secret key it gives away access to the model. Thus, there needs to be another third party server that controls access to the decrypted data and secret key, so the model owner cannot see the results as they are decrypted, and the leasing server does not get access to the private key.

One solution for this is using a Trusted Execution Environments (TEEs) such as IBM's Hyper Protect Virtual Server (HPVS) [6] or Intel's Secure Guard Extension (SGX) [7] as a decryption service, which enables a secure environment for these specific tasks, and controls who can authenticate and what is accessible to them. Note that while the TEE gets permission to see the secret key, it can only use it for the dedicated task of decrypting a given query, otherwise the leasing server can trick it to decrypt the encrypted model. In that sense, it is still a challenge to audit or verify that only legitimate queries are provided to the decryption service. Moreover, in order for the decryption service to not learn the decrypted results, some kind of encryption mask should be used in the process.

Even if we assume a semi-honest leasing server architecture that always provides legitimate queries, using another model is not a simple endeavor. It takes a large amount of infrastructure in order for it to function. There are environments that need to be set up for inference and environments set up for decryption. Specifically, different keys: public, private and evaluation, are needed to be in different places, and access will need to be restricted to the appropriate parties. Creation of these keys is also a concern, as well as where they are stored in order to achieve scalable and efficient solutions. Moreover, the size of a generative AI model is usually measured in GBs and even the simpler inference operation is required to run in less then several seconds. This may require dedicated hardware that clients do not possess.

In the second scenario, the model is uploaded in plaintext to a trusted environment, which provides it as a Machine Learning as a Service (MLaaS) solution to the users. To protect the input data, the users encrypt their queries using FHE and upload them to the MLaaS who performs the inference operation and returns the encrypted results to the user. Finally, the user decrypts the results to obtain the desired answer.

Also in the second scenario, key handling is a challenge. The key size of, for example, the evaluation keys can be more than several GB in size. Uploading them, even in an offline preparation stage, requires adequate storage location and it can be a non-trivial task when dealing with edge or low-end devices.

Listing all of the challenges that are needed to be overcome in order for an enterprise to host an FHE solution end to end for a generative AI task is out of the scope of this paper. For example, there are already many papers that deal with latency and accuracy issues of running an encrypted Deep Learning (DL) model e.g., [8, 9, 10]. In contrast, we identify that the challenge of handling the FHE keys and setup of the infrastructure to work at scale was left out from most modern papers, which usually assume that such a solution exists. For example, [11] defines the schemes and the keys but not how to use them, and a recent paper [12] defines the recommended parameters for these schemes but again without discussing the challenges of handling the keys. We claim that the above assumption is not the case and raise several issues that need to be addressed. To demonstrate these challenges we describe a simple machine learning solution using IBM's He4Cloud and the obstacles we encountered when we designed and set up the relevant infrastructure.

The paper is organized as follows. Section 2 reviews the current National Institute of Standards and Technology (NIST)'s Key Management System (KMS) standard Federal Information Processing Standards (FIPS) 800-57 and identify the gaps that need to be filled for it to capture HE-based KMS solutions. A description of a hierarchical KMS that can be implemented today using standard KMS services is provided in Section 3. We report on a concrete instantiation of HKMS in IBM's HPVS in Section 4. Finally, Section 5 discusses the issues that we encountered while designing and maintaining a basic FHE solution.

For brevity, Table 1 lists all the keys mentioned in this paper:

Table 1 – Legend of key names used in the paper

Key Name	Description
HE secret key (*sk*)	Private key used for HE decryption
HE public key (*pk*)	Public key used for HE encryption
HE evaluation key (*ek*)	Key enabling HE computations on ciphertexts, in some papers it is part of the public key
HE bootstrapping key (*bk*)	Key used to refresh ciphertexts in HE, in some papers it is part of the evaluation keys
HE rotation key (*rk*)	Key used to perform ciphertext rotations in HE, in some papers it is part of the evaluation keys
HE key switching key (*ksk*)	Key enabling switching ciphertexts encryption between keys
AES key encryption key (*kek*)	Key used to encrypt other AES keys
Envelope key (*ek*)	Key used to securely wrap (encrypt) other keys, similar to *kek*
AES key (k_{aes})	Symmetric key for AES encryption/decryption
KMS master key (*mk*)	Root key managing KMS operations
KMS data encryption key (*dk*)	Key used to encrypt data within the KMS
Envelope key (*ek*)	Root key managing KMS operations

2. FIPS 800-57 STANDARDIZATION GAPS

Cryptographic keys play an important role in cryptographic algorithms. Having unique and well-formatted keys is a prerequisite for the security guarantees provided by these cryptosystems. However, once an adversary puts its hands on these keys, the associated cryptographic scheme can no longer guarantee the confidentiality or the integrity of the key owner's data. Similar to many other solutions and libraries that provide cryptographic capabilities, the security guarantee that an FHE-based solution provides, depends on the security guarantees that the user's system can ensure for the generated cryptographic keys.

To ensure proper handling of keys throughout their lifecycle, standards organizations such as NIST provided KMS recommendations [13] and further recommend using them as part of their security requirements for cryptographic modules [14]. While these guidelines provide general recommendations, they do not necessarily capture issues and concerns that are specific to FHE. Note that there may be other KMS standards and we chose FIPS 800-57 as a representative case to show the gaps and the challenges, which we believe are not just semantic challenges.

NIST's FIPS 800-57 [13][Section 4] refers to three types of approved cryptographic algorithms: hash functions, symmetric-key algorithms, and asymmetric-key algorithms. Nevertheless, it does not explicitly mention FHE or refer to it. One possible reason is that until a decade ago, FHE was considered impractical, and only a small number of organizations have experimented with it. In fact, until recently, standard organizations such as NIST were asking the cryptographic community to focus on other types of cryptographic algorithms, such as lightweight cryptography [15], or post-quantum cryptography [16].

Recently, this situation has changed, and we see a proliferation of FHE solutions, e.g., [8, 9, 10]. Large companies such as IBM [17], Microsoft [18], Google [19], and Intel [20] see the potential of FHE solutions for the cloud and dedicate an increasing amount of resources to it. In addition, this change led standards organizations to put more focus on FHE, where some examples include NIST [21, 22], The Open Industry / Government / Academic Consortium to Advance Secure Computation [11], and ISO/IEC [23, 24][4]. While it may take time until we see a final standard, we can already prepare for that moment and examine the current KMS standards [29, 13] and identify the gaps that need to be complete.

FHE schemes are commonly built on top of some symmetric or asymmetric schemes. Therefore, we may see them partially supported by FIPS 800-57 [13]. Nevertheless, their keys do not fall under any of the keys defined in Section 5.1.1 of [13]. In FHE there are different sets of keys that can be created. To keep the discussion simple, multi-

[4] The old ISO standard draft [23] from 2021 was replaced by a new draft in 2024/5, which is divided into five parts. Specifically, these parts address: [24], general FHE concepts and principles, including foundational definitions, formats, and security models; [25], the BGV and BFV schemes, along with parameter selection for different security levels; [26], the CKKS scheme, with corresponding parameter guidance; [27], table-based FHE arithmetic mechanisms using look-up evaluation techniques and security-level parameters; and [28], scheme-switching mechanisms across BFV/BGV, CKKS, and CGGI. To the best of our knowledge, these drafts focus solely on describing keys as polynomials or other mathematical representations and do not address key management issues within KMSs.

party or multi-key FHE solutions are not discussed[5]. In its most basic form (assuming the workload needs the ability to multiply data), there are three keys that are used, a public key commonly used for encryption, an evaluation key for potential data processing, and a private key for decryption.

The first three sections of [13] include a general introduction to the glossary and terms that are used throughout the document and explanations about the different security services (e.g., confidentiality, data integrity, and authentication) a solution designer can use. The situation gets more complex in Section 4 that deals with cryptographic algorithms. At its beginning, we observe the first major gap:

> "FIPS-approved or NIST-recommended cryptographic algorithms shall be used whenever cryptographic services are required".

However, currently, no FHE algorithm is NIST approved. Subsequently, the standard specifies the three basic classes of approved algorithms: hash functions, symmetric-key algorithms, and asymmetric-key algorithms, which are defined according to the number of keys associated with each one of them. Unfortunately, FHE does not fall under these categories and as we saw it involves more keys than defined for the specified algorithms.

The main gap lies in Section 5, which classifies the different types of keys and provides recommendations for using them. It starts by referring to FIPS 800-133 [30] for the generation process of these keys, which also needs to be modified in a similar way to [13]. The only approved algorithms by [30][Section 3] are asymmetric-key algorithms and symmetric-key algorithms. In addition, [30][Section 5.3] discusses how an asymmetric cryptography key generation procedure should distribute its keys. In there, it emphasizes that the receiver of a static public key should verify it before using it e.g., an ITU-T X.509 certificate generated by the key owner. The situation is similar for the public and evaluation keys of an FHE scheme. An interesting question is whether ITU-T X.509 certificates can support the large size of these keys, or whether a new mechanism is required.

Furthermore, three types of keys are defined in [13][Section 5.1.1]: public, private, and (secret) symmetric keys.

[5] In the context of multi-party FHE, several approaches exist. In multi-key FHE (MK-FHE), each user possesses their own key, and decryption requires cooperation among parties. While encryption in MK-FHE resembles the single-key case, evaluation and decryption demand additional policy mechanisms that are beyond the scope of this work. Alternatively, threshold FHE involves all parties jointly generating a shared key, which is then used for encryption and requires an interactive protocol for decryption. Both MK-FHE and threshold FHE introduce distinct challenges that, for brevity, we leave outside the scope of this paper. Although we believe that solving these challenges is also critical to have collaboration-based solutions.

With FHE a new set of keys should be defined, specifically, we refer to evaluation, bootstrapping, and rotation keys, which we may group together under the wide definition of evaluation keys (see also Table 2). One unique property of such keys is that in many cases they endorse the associated FHE scheme with the property of *circular security*. The reason is that they encrypt a variant of the secret key. If in the future we will see an attack that shows how to extract the secret key from an evaluation key it can be devastating for the encrypted data. Thus, it is important to consider who has access to these keys even today. To this end, the list of key-types defined on [13][Section 5.1.1] should at least be extended to include:

- **Private data-encryption keys:** These keys can be used with (a)symmetric-key FHE algorithms to apply confidentiality protection to data (i.e., encrypt plaintext data). The same key is also used to remove the confidentiality protection (i.e., decrypt the ciphertext data). This key is not used for authentication primitives.
- **Public data-evaluation keys:** These keys are used with asymmetric-key FHE algorithms to evaluate a function on an untrusted platform over encrypted data.
- **Public data-encryption keys:** These keys are used with asymmetric-key FHE algorithms to allow untrusted parties to encrypt data that can only be decrypted using a matching secret key.

Cryptoperiod. [13][Section 5.3] defines and gives motivation for the term cryptoperiod, which is the lifetime of a specific key or the number of invocations of it within the context of a given algorithm. As mentioned in [13][Section 5.3.2], "short cryptoperiods enhance security". However, due to the latency of FHE operations which is often higher than other cryptosystems such as hash functions, symmetric, and asymmetric encryption, having a short cryptoperiod, which translates to frequent re-encryption of masses of data, can lead to high overheads on the KMS systems.

FIPS 800-57 [13][Section 5.3.3] lists several factors that affect cryptoperiods and states that keys that protect the confidentiality of communications are often shorter than keys that protect the confidentiality of stored data. FHE presents a new type of trade-off. The encrypted data is usually transmitted under a secure channel such as Transport Layer Security (TLS) 1.3 [31] and thus it cannot be treated as encryption in transit (unless a protocol such as MezzoTLS [32] is used). On the other hand, it is often the case that data should be used only once (e.g., a query) and deleted immediately after that. Thus, it cannot be considered an encryption at rest mechanism.

The situation might get even more complicated when considering, for example, the first scenario from the introduction. The model owner uses the secret/public key to encrypt its model and asks the users to encrypt

Table 2 – Comparison of standard and FHE-specific key types

Key Category	Description
Standard Keys (FIPS 800-57, Section 5.1.1)	
Public Key	Used for asymmetric encryption; publicly shared.
Private Key	Used for decryption/signing; must be kept secret.
Symmetric Key	Shared secret for symmetric encryption/decryption.
FHE-Specific Keys (not yet standardized)	
Evaluation Key	Enables homomorphic operations on ciphertexts.
Bootstrapping Key	Used to refresh ciphertexts by reducing noise.
Rotation Key	Enables rotation of encrypted vectors.
Key Switching Key	Transforming ciphertexts from one key to another.

their samples. In most cases, the users' samples contain more sensitive data than the AI model. The reason is that it reveals information that can directly be associated with a single person, whereas the model contains aggregated data about the training dataset. This suggests that the cryptoperiod should be determined based on the sensitivity level of the samples.

On the other hand, the model is stored on the leasing server for a long time, while the users' samples are only used during the inference period and are supposed to be erased immediately after the computation ends. Thus, according to [13][Section 5.3.3], the cryptoperiod should be determined based on the sensitivity of the model. Given the new key types, and the unique usage model of FHE-based systems, [13][Section 5.3.6] should be updated with new cryptoperiod recommendations for these keys, and [13][Section 6.1.1, Table 5] should also be updated accordingly.

FIPS 800-57 [13][sections 5.3.4 and 5.3.5] defines the cryptoperiod required for asymmetric and symmetric encryption, respectively. A subsection that defines the relations between the different cryptoperiods of FHE keys is therefore missing. To understand this relation, we consider Section 5.3.4, which states that a different cryptoperiod is defined for the secret and public keys of asymmetric encryption. Their lifetime starts at the same time, but the public key is allowed to be used, e.g., for verifying digital signatures much after the last time that the secret key is allowed to sign data.

In FHE, we have at least three types of keys, secret public and evaluation keys, where the latter can be broken into subcategories such as key-switching, rotation, and bootstrapping keys. The secret key is needed for decryption; without it, there is no purpose in using FHE. In contrast, the public key is required only for the encryption phase, and the evaluation keys are required only during the computations. Once the computations are over, the results can: a) be sent to the users for decryption using the secret key; b) stored and be decrypted at a later time; c) re-encrypted under a Hardware Secure Module (HSM), using a reversed hybrid re-encryption and stored under a symmetric encryption scheme.

Another challenge with FHE is its malleability. Assume that a standard defines that a public key cannot be used after a certain date. This does not mean that users cannot generate new ciphertexts (without the secret key). All a user should do is to capture a valid ciphertext c, reduce it from itself in a homomorphic way to get an encryption of 0: $c' = c - c = Enc(m - m) = Enc(0)$. Subsequently, the users can take any value v and add it to c' to get $c' + v = Enc(0) + v = Enc(v)$ as desired. Consequently, it is only meaningful to define a cryptoperiod for a public key if the users have no access to prior ciphertexts or are trusted to avoid encrypting new data when it is forbidden. The latter also means that the cryptoperiod depends on the refreshment rate of ciphertexts. Note that the above does not mean that an FHE scheme can be used without a public key, as the above mechanism is deterministic, which makes an FHE scheme not semantic secure.

Validity. FIPS 800-57 [13][Section 5.4.2] defines that before using any key the recipient of the key should assure the validity of the domain parameters; this can be done manually, by checking the key metadata, or if a cryptographic certificate is attached to the key it may include such an assurance. The exact process of how to ensure these parameters for FHE keys needs to be defined. For example, it should provide assurance that the scheme parameters provide the expected level of security, which is not trivial due to a large number of FHE configurations available today. It should also ensure that a key is not a weak key.

Key compromising implications. FIPS 800-57 [13][Section 5.5.1] describes the implications of compromising a key. While it clearly explains what can go wrong when a secret data encryption key is compromised, the implications of a compromised evaluation key are not defined. FHE schemes are malleable, which means that if an adversary gets access to the evaluation key, it can manipulate FHE ciphertexts and thus break their integrity. In that case, users should be alerted that they can no longer trust the validity of the computation.

Confidentiality of keys in transit. FIPS 800-57 [13][Section 6.2.1] defines the protection mechanisms for key information in transit. Specifically, Section 6.2.1.2 defines that confidentiality is required for symmetric keys and private keys in transit. Due to the malleability property of FHE schemes, we suggest also including the public and evaluation keys under this list.

The list above of challenges for the current KMS standard is just partial, and there are probably many more aspects that need to be addressed. Some examples, include the time to rotate keys and ciphertexts, how to handle key switching keys that encrypt other keys, how to handle transciphering keys etc. All of these challenges need to be addressed and an updated standard is required before we will see a broad use of the FHE technology for securing generative AI applications.

3. HIERARCHICAL KMS

Traditionally, to protect the keys that are used for encryption/decryption, an HSM is used. In an FHE workload environment, this should be no different. HSMs are typically known to manage all keys, keep them safe, and usually contain a special processor to make this process secure. A common pattern for an HSM is to store the key that is needed for encryption/decryption on the processor, or a key encrypting key, and the data is required to be moved to the HSM itself via an Application Interface (API) and payload. Most HSMs therefore are not expecting the massive amounts of data that would be required to be sent during an FHE workload session and therefore would not be able to be used with FHE keys due to the message size limitations. Another issue is that the key sizes for most FHE generated keys are larger than the limits of what the majority of current HSMs allow. Due to these issues not being resolved with the current HSM hardware (future generations will be different), a traditional HSM cannot be used off-the-shelf in the infrastructure.

Abstracting up a level an enterprise might use a KMS. This is a much broader implementation of a system for securing sensitive keys. It encompasses the generation, storage, usage, and replacement of keys, but it still needs to consider security. Many KMSs have HSMs underneath as the cryptographic processor of the information. While some KMS solutions can handle larger key sizes, there still remains the challenge of coding to the lowest common denominator, and systems without a cryptographic processor are inherently a less safe solution. Until standards are introduced detailing how these keys should be created, and stored, we will have this disconnect.

In this section, we present a KMS design that we call a Hierarchical Key Management System (HKMS). HKMS leverages current standard KMS/HSM solutions that provide symmetric encryption/decryption capabilities to a KMS, which in turn provides FHE capabilities.

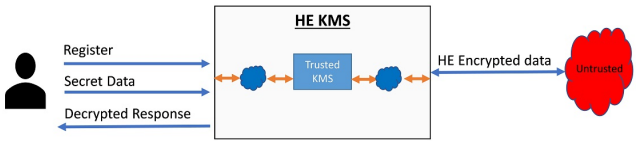


Figure 1 – An HKMS illustration showing two flows. First, the user registers with the HKMS, which generates the relevant HE keys. Subsequently, the user can upload secret data to the KMS and request encryption or decryption. The user can also ask the HE-KMS to transfer homomorphically encrypted data directly to an untrusted location.

Fig. 1 shows a high level illustration of the HKMS design. It provides at least the following three APIs: register, encryption, and decryption. A user uses the registration API to let the KMS know that a new set of keys is required. The KMS is responsible for associating the user credentials with the newly generated keys. We provide the exact details of how HKMS uses the trusted symmetrical KMS below. After successful registration, the user can use its credentials to identify itself to the KMS and perform encryption and decryption operations on its data using the generated key set. We stress that the HKMS does not need to be implemented at one location, i.e., at the cloud servers or the end-user side and that there are different ways to implement it. In Section 5 we will consider a HKMS that is implemented on the client premises.

Although HKMS provides encryption/decryption capabilities, it does not aim to provide a full Homomorphic Encryption as a Service (HEaaS) solution. As mentioned above, it is even better to separate those capabilities to reduce the size and scope of the KMS implementation and thereby reduce the potential attack surface.

Fig. 2 presents a possible HEaaS solution that uses HKMS. This solution can be implemented over several locations, e.g., some components can be implemented on the end-user side while others are implemented on the untrusted cloud site.

A user always starts by submitting a registration command that transparently goes to the HKMS. The HKMS generates the keys, which it can store locally in plaintext or encrypted at an external storage location. The user can then upload secret data or code, encode it using some FHE compiler, e.g., HElayers [17], and encrypt it using the KMS. After encryption, the data can be stored at a storage location or delivered directly to the FHE evaluation unit. In addition, the user can provide public input directly to the FHE evaluation unit, such as the desired model parameters or architecture. Finally, the results are either stored or moved to the KMS for decryption. The FHE decoding unit performs the final modifications

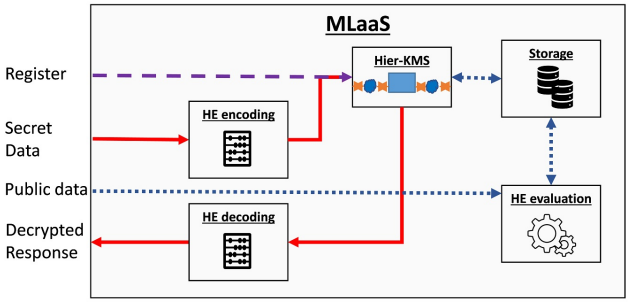


Figure 2 – An example design of an HEaaS solution. Red arrows indicate channels carrying unencrypted data. This "cleartext" data must first be encoded before encryption by an HE compiler such as HElayers to achieve optimized bandwidth usage. Since this data is unencrypted, it should be handled only by trusted components. Purple dashed arrows represent command signals, such as a command to register a new user in the HKMS system. When these commands include secret data they should be mounted under a secure channel. Finally, blue dotted arrows denote encrypted or public data, such as an encrypted ML model used for HE evaluation. It is assumed that all data in the storage area is either encrypted or not a secret.

required before providing the results to the user. Because the FHE encoding/decoding units handle unencrypted data they should use a secure transfer protocol such as TLS 1.3 [31] to transfer data from and to them, and in most cases, should be implemented on the end-user side.

3.1 Using a trusted KMS

We now further elaborate on the HKMS flows and explain how they leverage the internal traditional symmetric encryption KMS. We consider two main flows: registration and usage. The purpose of the registration flow is to ask HKMS to generate a new key and associate it with the user credentials. The flow assumes that it is possible to generate the FHE secret key in a deterministic way from a pseudo-random 256-bit seed [17]. The usage flow is a combination of the encryption and decryption flows, which for brevity we consider together. This allows us to easily demonstrate how an HKMS can fit into an HEaaS service. Fig. 3 presents the flows using an HKMS with three components: the trusted (standard) KMS, the untrusted storage location, and the HKMS main flow component. Panels a,b and panels c,d present the flows with and without *envelope encryption*, respectively. The input to the usage flows is the secret *data* and the output is m_{res}. Using the terms of Fig. 2 the input data arrives from the FHE encoding component while m_{res} is delivered to the FHE decoding unit.

Before we continue, we remark on the threat-model required by HKMS.

Remark 1 *The HKMS is an abstract design that admits multiple possible instantiations, each of which must account for its own specific threat landscape. This is analogous to standard KMS deployments, where different adversarial considerations arise depending on the deployment context. For example, an on-premise KMS may face insider threats, physical tampering, or key leakage due to misconfigured access controls. In contrast, a cloud-based KMS may be exposed to risks such as side-channel attacks in multi-tenant environments, unauthorized access by cloud administrators, or vulnerabilities in the virtualization layer. Accordingly, we strongly recommend that practitioners clearly define their threat model and system requirements before selecting or deploying an HKMS instantiation.*

3.1.1 Registration flow

The registration flow contains two subflows: 1) generating an encrypted initialization seed; 2) using the seed to generate FHE keys. The user triggers the flow by calling the HKMS registration API.

Generating an encrypted initialization seed.

1. HKMS generates a 256-bit seed s using an approved Cryptographic Random Number Generator (CRNG); see [33, 34].
2. HKMS opens a secure communication channel (e.g., TLS 1.3 [31]) with the trusted KMS system *KMS*.
3. HKMS sends s and the user's credentials *cred* to *KMS*.
4. *KMS* generates a new symmetric encryption (e.g., AES-256) key (k_{aes}) and associates it with *cred*. If such a key already exists *KMS* can reuse it.
5. *KMS* uses k_{aes} to encrypt s and returns the encryption blob $Es = \text{AES-GCM-ENC}_{k_{aes}}(s)$ to the user.

Note that in Step 3, it is possible to send sk instead of s and ask the KMS to encrypt sk. This may require extra bandwidth but will spare the need to regenerate sk from s in the next subflow.

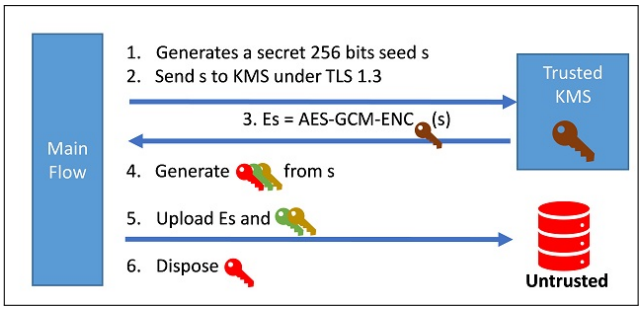
Using the seed to generate the FHE keys.

1. HKMS generates the secret sk, public pk, and evaluation keys ek from s.
2. HKMS uploads and stores (Es, pk, ek) possibly in an untrusted storage location. Access to these keys should be granted based on the threat model and *cred*.
3. HKMS wipes (s, Es, sk, pk, ek).

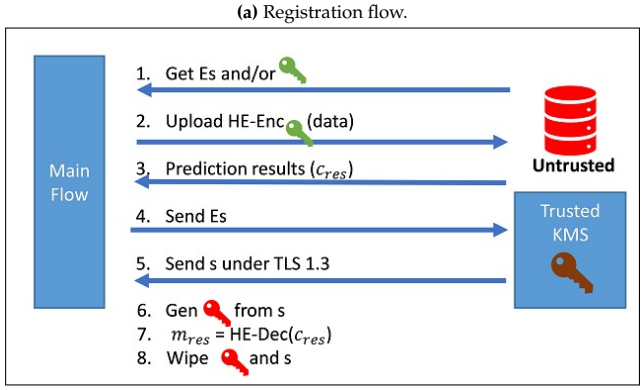
In solutions where there are many end users and all of them should have access to the secret key, it is possible, to send in Step 2 only (Es, ek) and prevent the need to store pk, which in many FHE schemes can be generated directly from sk.

3.1.2 Usage flow

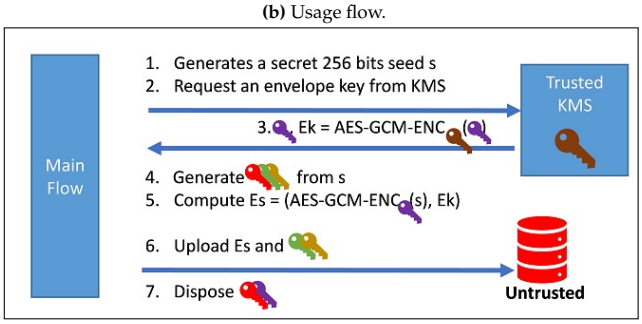
The usage flow combines two subflows: encryption and decryption.



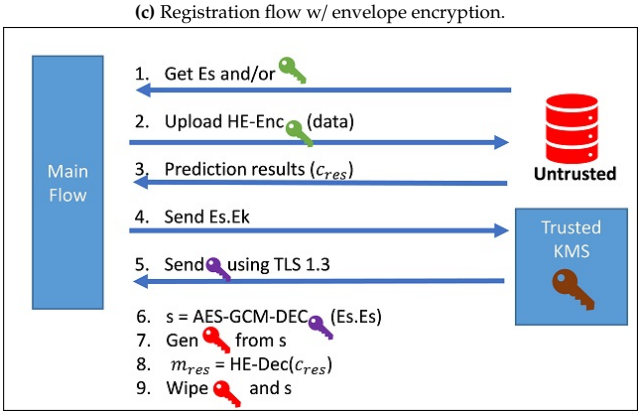
(a) Registration flow.



(b) Usage flow.



(c) Registration flow w/ envelope encryption.



(d) Usage flow w/ envelope encryption.

Figure 3 – Hierarchical-KMS registration and usage flows with and without envelope encryption. Red, green, yellow keys are the secret, public, and evaluation keys respectively. The brown key is the KMS AES master key, and the purple key is the envelope key. *s* is the seed, and *Es* stands for encrypted seed, *Ek* stands for envelope key. See text for more details.

Encryption.

1. HKMS asks the storage for *pk* for a given *cred*.
2. HKMS uses *pk* to encrypt some data $Ed = \text{Enc}_{pk}(data)$.
3. HKMS stores *Ed* in the storage location.

Subsequently, the HEaaS service performs some computations on the encrypted data, e.g., a classification model, and stores the encrypted output c_{res} in the storage location. Upon request (or immediately), it asks the HKMS to decrypt it using the following flow.

Decryption.

1. HKMS asks the storage for *Es* for a given *cred*.
2. HKMS opens a secure communication channel (e.g., TLS 1.3 [31]) with *KMS*.
3. HKMS sends *Es* to *KMS*.
4. *KMS* identifies the user's credentials and uses k_{aes} to decrypt *Es*, and returns the original seed $s = \text{AES-GCM-DEC}_{k_{aes}}(Es)$ to HKMS.
5. HKMS regenerate *sk* from *s*.
6. HKMS uses *sk* to decrypt the results $m_{res} = \text{Dec}_{sk}(c_{res})$
7. HKMS wipes (Es, s, sk, pk).

When a set of FHE keys is expected to be used only once (i.e., as an ephemeral key set), it is possible to unify the registration and usage flow by avoiding uploading *Es* to the cloud, i.e., using *sk* and deleting it after use. Using this approach avoids the benefit of caching the pre-generated keys, which introduces a new trade-off that depends on the key generation time and uploading time and bandwidth. When caching is still required, it is still possible to spare the first *sk* retrieval flow by deferring deleting it from the registration flow to the usage flow.

3.2 Using envelope encryption

The above registration and usage flows assume the trusted KMS can be trusted to see the key seed or the key itself. However, in many cases a solution designer would prefer to avoid this. To this end, we present the flows for an HKMS that uses envelope encryption (Fig. 3 panels c,d).

3.2.1 Registration flow

Generating an encrypted initialization seed.

1. HKMS generates a 256-bit seed *s* using an approved CRNG; see [33, 34].
2. HKMS asks *KMS* for a data key *dk* for a given *cred*.
3. *KMS* generates a new symmetric encryption (e.g., AES256) key (k_{aes}) and associates it with *cred*. If such a key already exists, *KMS* can reuse it.

4. *KMS* generates a new data key dk and sends (dk, Ek = AES-GCM-ENC$_{k_{aes}}$(dk)) to HKMS over a secure communication channel (e.g., TLS 1.3 [31]).
5. HKMS encrypts s by Es = AES-GCM-ENC$_{dk}$(s).
6. HKMS wipes dk.

Using the seed to generate the FHE keys.

1. HKMS generates the secret sk, public pk and evaluation keys ek from s.
2. HKMS uploads and stores (Es, Ek, pk, ek), possibly in an untrusted storage location. Access to these keys should be granted based on the threat model and *cred*.
3. HKMS wipes (s, Es, sk, pk, ek).

3.2.2 Usage flow

The encryption flow stays the same as above.

Decryption.

1. HKMS asks the storage for Es, Ek for a given *cred*.
2. HKMS opens a secure communication channel (e.g., TLS 1.3 [31]) with *KMS*.
3. HKMS sends Ek to *KMS*.
4. *KMS* identifies the user's credentials and uses k_{aes} to decrypt Ek and returns dk = AES-GCM-DEC$_{k_{aes}}$(Ek) to HKMS.
5. HKMS decrypts s by s = SYMDec$_{dk}$(Es).
6. HKMS regenerates sk from s.
7. HKMS uses sk to decrypt the results m_{res} = Dec$_{sk}$(c_{res})
8. HKMS wipes (s, sk, dk).

The big advantage of using HKMS is that it allows using current FHE with standard KMS systems, without the need to design a new KMS or HSM mechanisms. However, this design also has some drawbacks. For example, when the HKMS uses a "far" storage device, uploading large keys such as pk and ek can become costly. Nevertheless, this happens only once during the registration flow and when the entire design performs a relatively high number of encryption/decryption operations, the registration cost is less critical. Another possible disadvantage happens when the HKMS main flow is executed on the users' devices. In that case, the trust model should consider all the execution layers that have access to the secret key. It is, therefore, preferred to run the HKMS part inside a TEE such as IBM's HPVS [6], Intel's SGX [35], AWS's Nitro Enclave [36], or Azure Confidential Computing [37]. If these are not available, restrict the access to the HKMS execution code as much as possible, e.g., by providing only the minimal set of APIs (registration, encryption, and decryption).

4. AN INSTANTIATION USING IBM'S HPVS

In this section we consider how to realize an HKMS using IBM's infrastructure. We start by recalling that there are three keys to maintain. The public and evaluation keys are assumed to be moved around through the different pieces of infrastructure, the trusted and client *leasing server*. They will be generated in the IBM Software Development Kit (SDK) hosted in the trusted environment in the setup phase. In theory only the server that hosts the homomorphic computations will have access to the evaluation key. The public key will be used in any place where data needs to be encrypted. That leaves the secret key. The secret key is only needed when data has to be decrypted. This means that it can be isolated to one place, and in this case, it is set up within an IBM confidential computing environment using IBM's hyper protect services.

IBM HPVS stands for IBM hyper protect virtual servers. It provides confidential computing for the protection of data in use using hardware-based techniques. HPVS provides a TEE based on IBM Secure Execution for Linux. It protects data in use by workloads on the HPVS instance, even from privileged access. It is part of several offerings in the portfolio that can be combined and used together. One of the other offerings is Hyper Protect Crypto Services (HPCS). This is an as a Service (aaS) key management and encryption solution, which allows for full control over the keys used for data protection [38]. HPCS is a secure key repository for distributing and managing keys across a cloud environment. A main feature is a secure API to interact with the key management service to manage keys. The service is built on FIPS 140-2 Level 4 certified hardware and PKCS #11 is supported. Single-tenant dedicated HSM domains are fully controlled by the user, and IBM Cloud administrators have no access, which provides the highest security offered by any cloud provider in the industry.

When an IBM HPVS is instantiated, it also enables secure API access to HPCS, provided that service is enabled. IBM HPCS has many features, but at the heart of it is a secure cryptographic processor. The secret key is still too large for it to handle, but instead of the HPCS storing the key, it can generate the secret key from a pseudo-random 256-bit seed.

The seed is then protected using envelope encryption and is stored inside the HPCS. The key encrypting key never leaving the HSM. Using that seed, the IBM SDK can then generate a secret key when needed. When the SDK is done using the key, it gets zeroized, relying on the HSM to give it access to the seed to re-create the secret key. The big advantage of this method as shown above is that it allows for use of a standard KMS, with a traditional HSM, to be part of the homomorphic encrypted workflow.

Deployment to HPVS is meant to complement a container-based solution. It follows the best practices of deploying to a container platform with some slight additional security steps. Once an image is created and stored in an Image Container Repository (ICR), it can be added to the hyper protect runtime environment. The desired image must first be encrypted and a contract must be created as part of the user input. The contract is a definition file in the YAML format that is specific to the IBM HPVS instance. A user must create this file as a prerequisite for creating an instance, and after this file is created, it must be passed as an input when the IBM HPVS instance is created. A user cannot create an IBM HPVS instance without a valid contract [39]. The contract is where connections, settings and environment variables are defined, and because of that is it also encrypted before deployment. Once the image and contract are encrypted they are then uploaded to the hyper protect runtime environment and the bootloader decrypts the contract. It looks at each section by checking to make sure each piece has not been modified since the original signing upon encryption, and then decrypts and deploys based upon what was configured. A part of the boot process is to create an encrypted disk. It creates a unique root disk encryption key to ensure protection of the root disk. The boot process validates the root partition. If the hash of the root partition does not match, the boot process does not continue because it assumes that the image was modified before the boot [40].

Using both HPVS and HPCS, it is possible to instantiate an HKMS, which provides a reduced attack surface solution for handling secret keys, compared to just storing them in the application memory, as is commonly reported in the literature.

There are alternative ways of creating a secure key handling process. Using an HKMS solution is one way to create components that can help adopt FHE key management with a HSM or KMS. Another way of doing this is to rely on the TEE itself and build around its advantages and limitations. This involves creating several servers with strict policy rules implemented to assure that unauthorized users can't gain access. This method also requires a developer to implement the seed construction and envelope encryption that is available through our solution. The HKMS is designed to run alongside a TEE or similar technology. It partly acts as a translation layer for communication with a KMS. Referring to Fig. 4, it can be seen that the gateway (an HKMS implementation), is a way to route all commands to/from the different places. There are options if this was to be implemented on other cloud offerings. For example, on the Microsoft/Azure cloud, their equivalent offering is called Azure Key Vault Managed HSM. This could be used for key management and a TEE to get similar results. As for performance, that is undetermined with this hypothetical setup.

5. AN HE4CLOUD MLAAS DEMONSTRATION

We are now ready to combine all primitives together to one infrastructure.

Fig. 4 is a schematic view of a basic MLaaS solution that involves two parties and uses IBM's He4Cloud as the underlying FHE engine. The solution follows scenario two from the introduction. The model owner of a generative AI model trained a plaintext model and would not like to share it with company `comp`. Here, `comp` has many employees that want to enjoy the model of the model owner. The users want to maintain the privacy of their queries from the model owner and thus decide on using an FHE-based solution. Another factor in influencing this decision to outsource is that the users may not have a robust enough infrastructure to handle the expected additional computation by themselves. For example, they may require strong GPUs or even dedicated FHE hardware if the model would have been shipped to them in an encrypted format.

The interface of the users can be either through a web interface or some other application. However, working with a web browser may pose some limitations. For example, client web browsers can use WebAssembly to execute FHE flows, but this approach suffers from the following drawbacks: (a) WebAssembly has limited memory space of 4GB due to using 32-bit address space [41], which cannot scale for large models; (b) WebAssembly runs in browsers, which have a weak trust assumption for holding the secret key in memory; and (c) private FHE keys are distributed to multiple clients using multiple processes, making them difficult to manage using WebAssembly alone. Limitations such as (b) and (c) exist also when considering other application interfaces. Consequently, it is preferred to avoid the FHE computations in these applications and outsource them to another component, which we call `Gateway`.

The `Gateway` implements a HKMS as well as the encoding and decoding capabilities required for the solution. In fact, the `Gateway` deals with all aspects related to FHE and hides it from the users' interfaces. It is the `Gateway` task to connect and communicate with the model owner infrastructure and upload and download the encrypted data using either TLS 1.3 [31] or MezzoTLS [32].

The reasons for using the `Gateway` are therefore as follows.

1. **Better isolation.** Some applications, such as browsers, are not considered secure enough for holding secret keys in their memory.
2. **Limited attack surface.** It's limiting the attack surface by handling all the secret keys in one location, in contrast to having a secret key shared between many users.

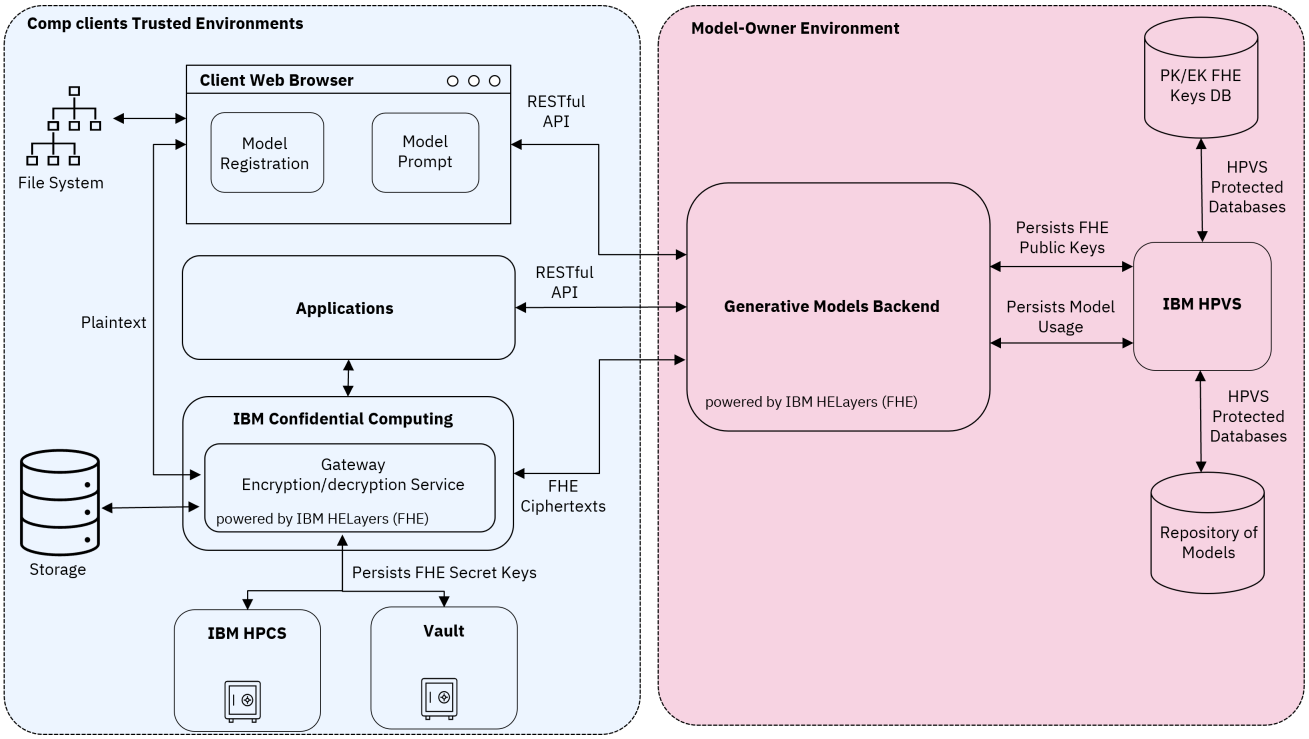


Figure 4 – A schematic illustration of the He4Cloud solution using a gateway as the HKMS built on top of IBM's HPVS and HPCS.

3. **Pre-fetching.** It avoids the expensive task of regenerating the evaluation keys (limitation (C) above). These, can be generated once and maybe even in advance to any query for multiple users. Moreover, the keys can be sent ahead of time (in an offline step) to the model owner server who can store them for a later use. Later, the keys can be identified using some credentials that were allocated by the `Gateway`.
   Note that both `comp`'s `Gateway` and the component that handles the keys at the model owner infrastructure reside within IBM's HPVS, as described in Section 4, which guarantees the integrity of the operation while also providing authentication and authorization capabilities for using the relevant keys. While the model-owner only handles "public keys" and thus only uses the IBM's HPVS for integrity of the computation, the `Gateway` also needs to handle the secret keys and thus it also uses IBM's HPCS. As an alternative, Fig. 4 also suggests using Vault instead of HPCS as another solution.
4. **Scalability.** The `Gateway` can be scaled horizontally and vertically.
5. **Bandwidth allocation.** The `Gateway` allows placing the components in different locations of `comp` infrastructure. For example, we can set the Gateway to have a much higher bandwidth allocation compared to the bandwidth permitted per every single user.
6. **Authorization.** The `Gateway` can serve as a key inventory with policies for different users allowing access to the FHE keys according to rules in the organization or groups.
7. **Encapsulation.** Encapsulates all the FHE knowledge and implementations in one place. This includes the configuration parameters which can be hidden from the user.

Table 3 presents a detailed comparison between the client-based WebAssembly and gateway-based architectures in the context of HE4Cloud, highlighting key differences in memory usage, security, scalability, and operational efficiency. The comparison demonstrates that the gateway-based architecture offers significant advantages, particularly in terms of scalability and security.

Fig. 4 shows that there are many more engineering considerations for designing a secure solution when using FHE, than just assuming that the user generates the keys and stores them somehow. Indeed, the above solution is not perfect, as currently no HSM for FHE is available, and we do not expect to see one before a standard is defined. But this is exactly the issue that this paper aims to address. Designing, developing, and manufacturing such HSMs can take several years, and investing in a specific design can turn out to be wrong if a future standard provides different recommendations. On the other hand, waiting for the standard to be ready can delay the generation of such HSMs by several more years. This is why we recommend starting the standardization process of KMS solutions as soon as possible to enable secure GEN AI solutions in the not-too-distant future, while also

Table 3 – Comparison of client-based WebAssembly and gateway-based architectures in HE4Cloud.

Aspect	Client-Based WebAssembly Solution	Gateway-Based Solution
Memory	Limited to 4 GB by 32-bit addressing.	No constraints; server-grade resources.
Keys security	Stored in untrusted client memory.	Secured in trusted server (HPCS, Vault).
Key management	Hard to manage across clients.	Centralized, policy-based control.
Eval Key Handling	Regenerated per client; inefficient.	Pre-fetched, reused, efficiently shared.
Attack Surface	High exposure: keys are distributed across many clients, increasing risk of compromise.	Minimized exposure: centralized key handling reduces potential attack vectors.
Encapsulation	FHE logic exposed to clients.	FHE logic hidden and centralized.
Scalability	Limited by client capabilities.	Scales horizontally and vertically.
Bandwidth	Limited per user; hard to optimize across sessions.	Centralized control enables higher throughput and optimized allocation.
Authorization	Hard to enforce fine-grained access.	Centralized access control and auditing.
HPVS Integration	Not applicable or limited.	Full integration with HPVS and optionally HPCS.

describing, designing, and testing KMS solutions, as in this paper, even before a standard arrives, to further learn the limitation and capabilities of the FHE technology.

Fig. 5 illustrates an optional deployment of the HE4Cloud system within IBM Cloud, leveraging the following managed services:

1. **IBM HPCS (KMS):** used to securely manage and protect FHE secret and evaluation keys.
2. **IBM Cloud Internet services (WAF):** provides application-layer protection against threats such as SQL injection, cross-Site Scripting (XSS), and DDoS attacks.
3. **IBM App ID:** enables user authentication, identity management, and secure access control without requiring custom authentication infrastructure.
4. **IBM Red Hat OpenShift:** used to deploy, orchestrate, and manage containerized services, ensuring scalable and secure integration of HE4Cloud backends within IBM Cloud.
5. **IBM Cloud load balancer:** used to route and balance traffic across service instances, enhancing availability, reliability, and performance.
6. **IBM Cloud object storage:** used to persist FHE public keys and deployed model artifacts in a durable and scalable manner.
7. **IBM Cloudant:** used to store metadata and usage information related to HE4Cloud operations.

The HE4Cloud framework supports multi-user deployment of machine learning models. Given the high computational overhead associated with FHE operations, it is essential for the system to be scalable. To address this challenge in a multi-user environment, HE4Cloud supports both horizontal and vertical scaling strategies. As illustrated in Fig. 5, the system can scale horizontally by adding additional machines or instances behind a load balancer to distribute the workload. Alternatively, it can scale vertically by increasing the computational resources, such as CPU and memory, of existing machines. These scaling mechanisms ensure that the system can efficiently accommodate growing demand while maintaining secure and efficient model execution.

6. PERFORMANCE EVALUATION

To give a sense of the runtime and network communication required by HKMS, Table 4 presents a summary of two deployed neural network models (Fraud Detection [42] and Heart Disease [43]) in the HE4Cloud framework, which integrates the HashiCorp Vault Transit Secrets Engine for secure key handling. In this setup, the transit engine is used to encrypt and decrypt the seed value that generates the FHE secret key. The table includes the total prediction time, key generation, seed vault operations, model encryption, and sample processing. It also reports the sizes of cryptographic artifacts and the input data shape, providing a clear view of the runtime and cryptographic overhead involved in secure model execution. Note that the time spent on seed encryption and decryption is very small and has minimal impact on the overall runtime. The system features 48 physical CPU cores (Intel® Xeon® Platinum 8260) and 188 GiB of RAM.

Table 4 – Runtime and cryptographic metrics for encrypted neural network inference in HE4Cloud

Metric	Fraud [42] Detection	Heart [43] Disease
Samples Encryption (s)	0.07	0.03
Prediction Duration (s)	0.44	0.17
Key Creation (s)	0.52	0.11
Predictions Dec. (ms)	3.00	1.00
Seed Vault Enc. (ms)	9.00	4.00
Seed Vault Dec. (ms)	5.00	4.00
Public Key Size (MB)	2.00	0.63
Secret Key Size (MB)	0.92	0.25
Seed Size (Bytes)	91	91

7. CONCLUSION

Keys are the core of every cryptographic function, and thus protecting them is crucial. Indeed, in recent years, we have observed major improvements in the characteristics of secure inference solutions for both classification and generation tasks. However, the focus has been on improving latency and accuracy, while leaving key han-

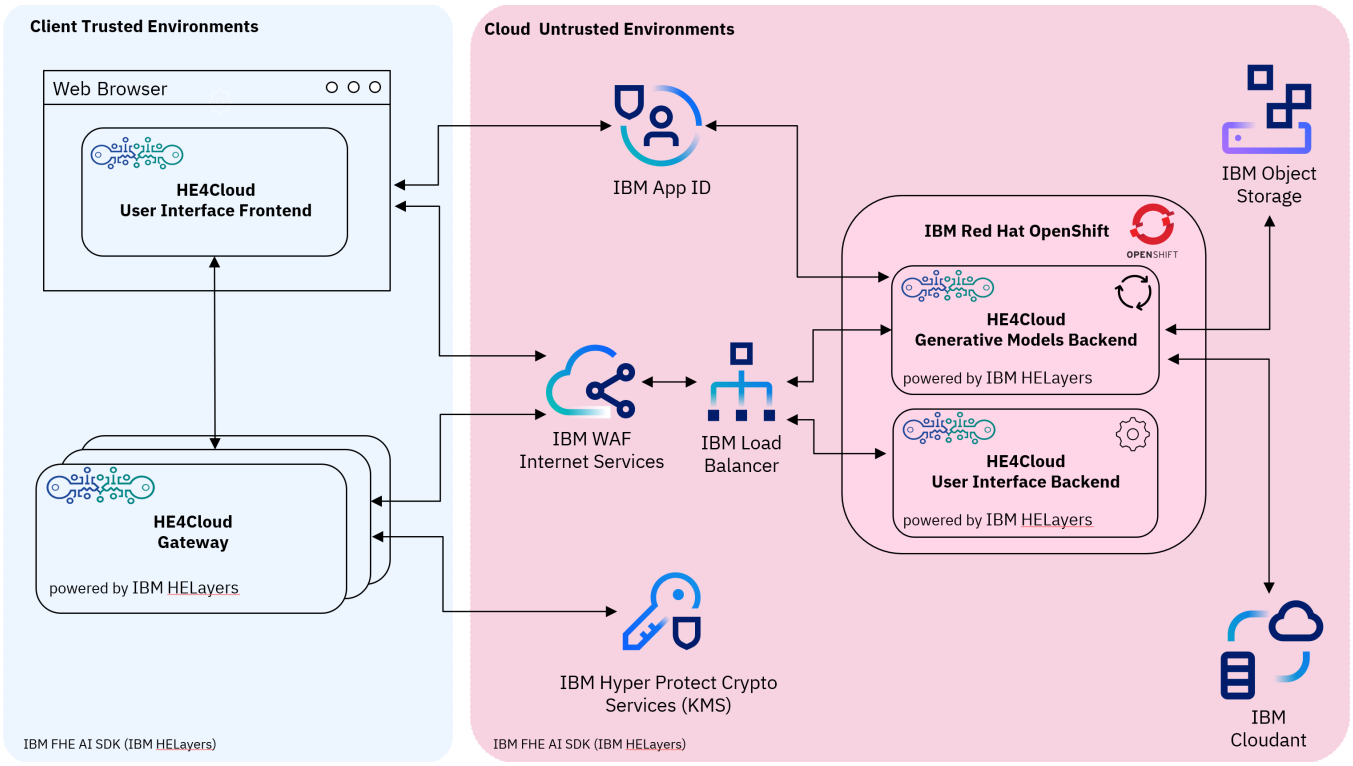


Figure 5 – An illustration of the HE4Cloud deployment in IBM Cloud.

dling to the implementers. We believe we have reached a point where the challenge of key handling should be further investigated in order to enable the broad adoption of FHE technologies.

In this paper, we highlighted these gaps to start the discussion. Furthermore, we explore solutions such as HKMS that can be deployed today, even though KMS standards (or even FHE standards) do not yet exist. Finally, we demonstrate how to instantiate HKMS within an IBM He4cloud solution using technologies such as IBM's HPVS and HPCS. We hope that the challenges discussed will be addressed by the community, alongside closing the latency and accuracy gaps, to achieve secure generative AI solutions.

A. A PSEUDO-CODE EXAMPLE

The following is a pseudo-code example for running a HKMS.

```
INITIALIZE encryption requirements (he_run_req)
CONFig. encryption context

INITIALIZE model hyperparameters
LOAD plain model from MODEL_JSON and MODEL_H5
COMPILE model to get encryption profile

CREATE encryption context from profile
INITIALIZE public functions with encryption enabled
SAVE public key to buffer
SAVE secret key and seed

ENCRYPT seed using KMS
CREATE empty encrypted model using context
ENCODE and ENCRYPT model
SAVE encrypted model to buffer

LOAD input samples and labels
INITIALIZE model input/output encoder
LOAD server-side encryption context using public key
DECRYPT seed using KMS

LOAD secret key into server context
INITIALIZE encrypted data container for samples
ENCODE and ENCRYPT input samples

INITIALIZE encrypted data container for predictions
RUN encrypted prediction
DECRYPT and DECODE predictions
```

REFERENCES

[1] Shai Halevi. "Homomorphic Encryption". In: *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*. Ed. by Yehuda Lindell. Cham: Springer International Publishing, 2017, pp. 219–276. ISBN: 978-3-319-57048-8. DOI: 10.1007/978-3-319-57048-8_5.

[2] Zvika Brakerski, Craig Gentry, and Shai Halevi. "Packed Ciphertexts in LWE-based Homomorphic Encryption". In: *Public-Key Cryptography - PKC 2013*. Vol. 7778. 2013, p. 1. DOI: 10.1007/978-3-642-36362-7_1.

[3] Junfeng Fan and Frederik Vercauteren. "Somewhat Practical Fully Homomorphic Encryption". In: *Cryptology ePrint Archive* (2012). URL: https://eprint.iacr.org/2012/144.

[4] Zvika Brakerski. "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP". In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417 LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 868–886. ISBN: 978-3-642-32009-5. DOI: 10.1007/978-3-642-32009-5_50.

[5] Jung Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. "Homomorphic Encryption for Arithmetic of Approximate Numbers". In: *Proceedings of Advances in Cryptology - ASIACRYPT 2017*. Springer Cham, Nov. 2017, pp. 409–437. ISBN: 978-3-319-70693-1. DOI: 10.1007/978-3-319-70694-8_15.

[6] IBM. *IBM Hyper Protect Virtual Servers*. IBM Cloud service. Accessed: 2025-03-02. 2025. URL: https://www.ibm.com/cloud/hyper-protect-virtual-servers.

[7] Nir Drucker and Shay Gueron. "Achieving trustworthy Homomorphic Encryption by combining it with a Trusted Execution Environment". In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* 9.1 (Mar. 2018), pp. 86–99. DOI: 10.22667/JOWUA.2018.03.31.086.

[8] Moran Baruch, Nir Drucker, Gilad Ezov, Yoav Goldberg, Eyal Kushnir, Jenny Lerner, Omri Soceanu, and Itamar Zimerman. "Polynomial Adaptation of Large-Scale CNNs for Homomorphic Encryption-Based Secure Inference". In: *Cyber Security, Cryptology, and Machine Learning*. Ed. by Shlomi Dolev, Michael Elhadad, Mirosław Kutyłowski, and Giuseppe Persiano. Cham: Springer Nature Switzerland, 2025, pp. 3–25. ISBN: 978-3-031-76934-4. DOI: 10.1007/978-3-031-76934-4_1.

[9] Itamar Zimerman, Allon Adir, Ehud Aharoni, Matan Avitan, Moran Baruch, Nir Drucker, Jenny Lerner, Ramy Masalha, Reut Meiri, and Omri Soceanu. "Power-softmax: Towards secure llm inference over encrypted data". In: *arXiv preprint arXiv:2410.09457* (2024). URL: https://arxiv.org/abs/2410.09457.

[10] Itamar Zimerman, Moran Baruch, Nir Drucker, Gilad Ezov, Omri Soceanu, and Lior Wolf. "Converting transformers to polynomial form for secure inference over homomorphic encryption". In: *Proceedings of the 41st International Conference on Machine Learning*. ICML'24. Vienna, Austria: JMLR.org, 2024. DOI: 10.5555/3692070.3694670.

[11] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. *Homomorphic Encryption Security Standard*. Tech. rep. Toronto, Canada: HomomorphicEncryption.org, Nov. 2018. URL: https://homomorphicencryption.org/standard/.

[12] Jean-Philippe Bossuat, Rosario Cammarota, Ilaria Chillotti, Benjamin R. Curtis, Wei Dai, Huijing Gong, Erin Hales, Duhyeong Kim, Bryan Kumara, Changmin Lee, Xianhui Lu, Carsten Maple, Alberto Pedrouzo-Ulloa, Rachel Player, Yuriy Polyakov, Luis Antonio Ruiz Lopez, Yongsoo Song, and Donggeon Yhee. *Security Guidelines for Implementing Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2024/463. 2024. URL: https://eprint.iacr.org/2024/463.

[13] Barker Elaine. "NIST Special Publication 800-57: Recommendation for Key Management Part 1 – General". In: (2021). DOI: 10.6028/NIST.SP.800-57pt1r5.

[14] NIST. *FIPS PUB 140-2: Security Requirements for Cryptographic Modules*. 2002. DOI: 10.6028/NIST.FIPS.140-2.

[15] NIST. *Lightweight Cryptography*. https://csrc.nist.gov/projects/lightweight-cryptography. Last accessed 02 March 2025. 2023.

[16] NIST. *Post-Quantum Cryptography*. https://csrc.nist.gov/projects/post-quantum-cryptography. Last accessed 02 March 2025. 2024.

[17] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkowich, Dov Murik, et al. "HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data". In: *Proceedings on Privacy Enhancing Technologies* 2023 (1 2023), pp. 325–342. DOI: 10.56553/popets-2023-0020.

[18] Kim Laine. *Simple Encrypted Arithmetic Library 2.3.1*. Tech. rep. WA, USA: Microsoft, 2017. URL: https://www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf.

[19] HEIR Contributors. *HEIR: Homomorphic Encryption Intermediate Representation*. https://github.com/google/heir. 2023.

[20] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D.M. de Souza, and Vinodh Gopal. "Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52". In: *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 57–62. ISBN: 9781450386562. DOI: 10.1145/3474366.3486926.

[21] NIST. *Toward a PEC Use-Case Suite*. 2021. URL: https://csrc.nist.gov/CSRC/media/Projects/pec/documents/suite-draft1.pdf.

[22] NIST. *Privacy-Enhancing Cryptography (PEC)*. https://csrc.nist.gov/projects/pec. Last accessed 02 March 2025. 2025.

[23] ISO/IEC. *ISO/IEC 18033-6:2019 IT Security techniques — Encryption algorithms — Part 6: Homomorphic encryption*. 2021. URL: https://www.iso.org/standard/67740.html.

[24] ISO/IEC. *ISO/IEC CD 28033-1:Information security — Fully homomorphic encryption*. 2024. URL: https://www.iso.org/standard/87638.html.

[25] ISO/IEC. *ISO/IEC CD 28033-2:Mechanisms for exact arithmetic on modular integers*. 2024. URL: https://www.iso.org/standard/87639.html.

[26] ISO/IEC. *ISO/IEC CD 28033-3:Mechanisms for arithmetic on approximate numbers*. 2024. URL: https://www.iso.org/standard/87640.html.

[27] ISO/IEC. *ISO/IEC CD 28033-4:Mechanisms for arithmetic based on look-up table evaluation*. 2024. URL: https://www.iso.org/standard/87641.html.

[28] ISO/IEC. *ISO/IEC CD 28033-5:Mechanisms for Scheme Switching*. 2024. URL: https://www.iso.org/standard/87638.html.

[29] Barker Elaine, Smid Miles, Branstad Dennis, and Chokhani Santosh. "NIST Special Publication 800-130: A Framework for Designing Cryptographic Key Management Systems". In: (2013). DOI: 10.6028/NIST.SP.800-130.

[30] Barker Elaine, Roginsky Allen, and Richard Davis. "NIST Special Publication 800-133, Rev 2: Recommendation for Cryptographic Key Generation". In: (2020). DOI: 10.6028/NIST.SP.800-133r2.

[31] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446.

[32] Nir Drucker and Shay Gueron. "Mezzo TLS 1.3 Protocol, Suitable for Transmitting Already-Encrypted Data". In: *Cyber Security, Cryptology, and Machine Learning*. Ed. by Shlomi Dolev, Michael Elhadad, Mirosław Kutyłowski, and Giuseppe Persiano. Cham: Springer Nature Switzerland, 2025, pp. 92–99. ISBN: 978-3-031-76934-4. DOI: 10.1007/978-3-031-76934-4_6.

[33] Elaine B. Barker and John M. Kelsey. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. 2015. DOI: 10.6028/NIST.SP.800-90Ar1.

[34] Elaine Barker and John Kelsey. *Recommendation for Random Bit Generator (RBG) Constructions*. 2012. DOI: 10.6028/NIST.SP.800-90C.4pd.

[35] Intel ®. *Intel ® Software Guard Extensions Programming Reference*. https://software.intel.com/en-us/isa-extensions/intel-sgx. Oct. 2014.

[36] Amazon Web Services. *AWS Nitro Enclaves – Isolated Compute Environments for Confidential Computing*. https://aws.amazon.com/ec2/nitro/nitro-enclaves/. Accessed: 2025-06-26. 2021.

[37] Microsoft Azure. *Azure Confidential Computing*. https://azure.microsoft.com/en-us/solutions/confidential-compute/. Accessed: 2025-06-26. 2021.

[38] Greg Boland and Timo Kußmaul. *Get started with IBM HElayers and IBM Hyper Protect Virtual Servers for VPC*. IBM Developer Blog. Accessed: 2025-03-01. 2023. URL: https://developer.ibm.com/tutorials/awb-get-started-ibm-helayers-sdk-hyper-protect-virtual-servers-vpc/.

[39] IBM. *About the contract*. Accessed: 2025-03-03. URL: https://www.ibm.com/docs/en/hpvs/2.1.x?topic=servers-about-contract.

[40] IBM. *Attestation*. Accessed: 2025-03-03. URL: https://www.ibm.com/docs/en/hpvs/2.1.x?topic=servers-attestation.

[41] Ulan. *Proposal: Configurable Wasm Heap Limit*. Internet Computer Developer Forum. Discussing and community-consideration forum post. Jan. 2023. URL: https://forum.dfinity.org/t/proposal-configurable-wasm-heap-limit/17794.

[42] IBM HElayers. *Credit card fraud detection using NN inference*. Accessed: 2025-06-26. URL: https://ibm.github.io/helayers/user/resources.html#credit-card-fraud-detection-using-nn-inference.

[43] IBM HElayers. *Heart disease detection using NN inference*. Accessed: 2025-06-26. URL: https://ibm.github.io/helayers/user/resources.html#heart-disease-detection-using-nn-inference.

AUTHORS

Akram Bitar is currently with IBM Research in Haifa, Israel. Akram holds an M.Sc. in mathematics and computer science from the University of Haifa. Akram has broad professional experience in complex software systems architecture, design and development, of which he has a few years as a project leader. In addition, he has a strong technology background in FHE, data quality, syntactic data generation, storage systems, distributed systems, and computer networking.

Greg Boland is currently with IBM USA, a software engineer specializing in cryptographic firmware development, with a strong passion for creating innovative products and actively exploring new possibilities. His recent work has centered on enabling and educating others on Fully Homomorphic Encryption (FHE) for IBM Z systems. Greg continuously expands his expertise in integrating hardware and software cryptography techniques to ensure the security of data. He is driven by the ever-evolving landscape of emerging technologies and the constant need to address evolving security threats.

Nir Drucker is currently with IBM Research in Haifa, Israel. Nir holds a Ph.D. in applied mathematics (cryptography) from the University of Haifa and an M.Sc. in operations research from the faculty of Industrial Engineering and Management at the Technion I.I.T. FHE worked three and a half years as a senior applied scientist at AWS (cryptography) and eight years as a software developer at Intel. Nir's research interests focus on applied cryptography and applied security.