IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–30. DOI:XXXXXXXX

A Lattice-based Publish-Subscribe Communication Protocol using Accelerated Homomorphic Encryption Primitives

Anes Abdennebi[1] and Erkay Savaş[2]

[1] Department of Computer Science and Engineering, Sabancı University,
anesabdennebi@sabanciuniv.edu
[2] Department of Computer Science and Engineering, Sabancı University,
erkays@sabanciuniv.edu

Abstract. Key-policy attribute-based encryption scheme (KP-ABE) uses a set of attributes as public keys for encryption. It allows homomorphic evaluation of ciphertext into another ciphertext of the same message, which can be decrypted if a certain access policy based on the attributes is satisfied. A lattice-based KP-ABE scheme is reported in several works in the literature, and its software implementation is available in an open-source library called PALISADE. However, as the cryptographic primitives in KP-ABE are overly involved, non-trivial hardware acceleration is needed for its adoption in practical applications.

In this work, we provide GPU-based algorithms for accelerating KP-ABE encryption and homomorphic evaluation functions seamlessly integrated into the open-source library with minor additional build changes needed to run the GPU kernels. Using GPU algorithms, we perform both homomorphic encryption and homomorphic evaluation operations 2.1× and 13.2× faster than the CPU implementations reported in the literature on an Intel i9, respectively. Furthermore, our implementation supports up to 128 attributes for encryption and homomorphic evaluation with fixed and changing access policies. Unlike the reported GPU-based homomorphic operations in the literature, which support only up to 32 attributes and give estimations for a higher number of attributes.

We also propose a GPU-based KP-ABE scheme for publish/subscribe messaging applications, in which end-to-end security of the messages is guaranteed. Here, while the exchanged messages are encrypted with as many as 128 attributes by publishers, fewer attributes are needed for homomorphic evaluation. Our fast and memory-efficient GPU implementations of KP-ABE encryption and homomorphic evaluation operations demonstrate that the KP-ABE scheme can be used for practicable publish/subscribe messaging applications.

Keywords: Lattice-based cryptography · Attribute-based encryption · RLWE · PALISADE · GPU · Publish/Subscribe

1 Introduction

The attribute-based encryption (ABE) is introduced first in [1], where the correct decryption of ciphertext can be performed by users that have a certain set of attributes. There are two types of ABE, namely *ciphertext-policy attribute-based encryption (CP-ABE)* and the *key-policy attribute-based encryption (KP-ABE)*. In CP-ABE [8–15], as the access policy is defined over ciphertext, it must be known at the time of encryption. Therefore, the publisher has to know all access policies requested by the subscribers beforehand, which is not exactly suitable for the loosely coupled nature of publishers and subscribers in the

publish/subscribe (Pub/Sub) communication model. In the KP-ABE scheme [2–7], on the other hand, the plaintext is encrypted using a set of attributes as the public key and decryption keys are generated for access policies defined over a subset of the attributes. An access policy can be defined after the encryption, and publishers need to know neither subscribers nor the access policies improving subscribers' privacy. Also, the attributes can be the properties of the message to be encrypted, not the attributes of the recipients. Once the access policy is known, the original ciphertext is homomorphically evaluated and transformed into another ciphertext under different access policies by a third party, which can be a communication server that does not need to know a secret key. Therefore, KP-ABE stands as a suitable cryptographic scheme that provides all the primitives needed to ensure end-to-end security for Pub/Sub model.

In a pub-sub communication model with KP-ABE, we envision a communication service run by a third party, which provides long-term storage for, homomorphically evaluates, and forward ciphertexts to interested subscribers. Publishers and subscribers agree on a set of attributes of the messages, over which the policies will be defined and the attributes are not necessarily revealed to the communication service as they may be sensitive. The set of attributes is referred to as the dictionary, and its elements can be represented with pseudonyms whose mappings to real attributes are only known by publishers and subscribers. The subscribers register with the Pub/Sub service indicating the published messages they are interested in using an access policy defined over the attributes of the message. For instance, attributes can be keywords or categories of the messages, and the policy can be defined as messages containing certain keywords or falling into certain categories. In addition, the communication service can support search operations based on the set of attributes over the previously published messages. In the Pub/Sub scenario, the number of keywords or categories can easily reach thousands while their multitude in the access policies stays relatively low. However, as the number of policies for the same message can vary depending on the requesting subscribers, homomorphic evaluation can still be a computational bottleneck. Therefore, encrypting messages and homomorphic evaluations of ciphertexts and public keys can be time-consuming and call for non-trivial acceleration.

Dai et al. [20] report on a GPU accelerator of the KP-ABE scheme in [16] for PALISADE library [21] using a fast implementation of *Number Theoretic Transform (NTT)* operations in [17]. The GPU implementation of NTT in [17] uses a special form "carrier prime" which supports fast modular arithmetic. Nonetheless, using carrier prime leads to working with many smaller primes in RNS arithmetic [18] used for multi-precision integers to support large modulus in homomorphic encryption systems as well as KP-ABE. Also, the authors in [20] employ various other techniques to accelerate KP-ABE operations, such as *Non-Adjacent Form (NAF)* for slowing the noise growth in a zero-centered range such that the ciphertexts are decrypted successfully. Nevertheless, the implementation in the `PALISADE` library and its GPU accelerator still suffer from a high execution time due to overly involved encryption and homomorphic evaluation operations. For instance, on an Intel Core i7 processor for 8 attributes, homomorphic evaluation, and encryption functions take up to 3.4 seconds and 259 ms, respectively (Table II in [20]).

Another work [25] employs a technique known as subgaussian gadget decomposition, for which larger base $\boldsymbol{b}$ values can be used for the decomposition operation while in the original implementation in [21] and its GPU implementation in [20], $\boldsymbol{b} = 2$ is used for decomposition. While using larger decomposition bases can adversely affect the noise growth in the ciphertext, it significantly accelerates both the KP-ABE encryption and ciphertext homomorphic evaluation operations. Indeed, for eight attributes, the implementation in [25] performs homomorphic evaluation 18.3 times faster than the implementation in [20]. Nonetheless, when the number of attributes increases, even the fast method in [25] results in prohibitively high execution times. For instance, when the number of attributes $\ell = 32$,

the homomorphic evaluation operation takes as high as 5.67 s.

To the best of our knowledge, the implementation in [20] is the only work that introduces the GPU implementation of the *KP-ABE* scheme. Although it provides considerable acceleration, it has several shortcomings. As the attribute number, ring dimension, and modulus size increase, the timings become prohibitively high for any practical application of KP-ABE. First and foremost, the NTT algorithm employed and its GPU implementation and the use of RNS arithmetic are inefficient. Moreover, as the memory requirements also increase, the device memory (with the GPU used in [20]) becomes insufficient when the number of attributes exceeds 32. As a result of this, Dai et al. [20] provide estimations for execution times beyond 32 attributes. Moreover, it does not exploit the much faster KP-ABE algorithms introduced [25]. Finally, the accelerator in [20] is a separate GPU implementation and not fully integrated with the PALISADE library that implements other KP-ABE operations, which needs no acceleration and runs efficiently on a common CPU. Consequently, the GPU implementation falls short of being a real accelerator as its integration with a software library is non-trivial.

High execution times, especially for a moderately high number of attributes, fail to motivate users, researchers, or developers to integrate the KP-ABE scheme in practical applications, albeit with its highly advanced security properties. This work is a renewed attempt to address the implementation challenges of the KP-ABE scheme using improved algorithms and implementation techniques suitable for GPU devices.

Our Contribution. This work aims to provide an accelerated GPU implementation of the three most time-consuming KP-ABE operations when it is used to provide end-to-end security in Pub/Sub communication system. KP-ABE consists of five cryptographic functions: i) key generation ii) encryption, iii) Homomorphic evaluation of ciphertext, `EvalCT`, iv) Homomorphic evaluation of public key, `EvalPK` and v) decryption. As decryption and key generation operations are relatively fast and key generation is infrequently applied, we focus on encryption and homomorphic evaluation operations.

When a subscriber in Pub/Sub system forms an attribute-based access policy, `EvalPK` operation is applied to the attributes, and a new public key is obtained to compute a secret key for the access policy. Provided an access policy changes occasionally (fixed policy), `EvalPK` operation is not executed often. `EvalCT` operation needs to be performed every time a new message is published. When the policies are changing frequently, both operations are performed (changing policy). Similarly, publishers need to encrypt every message sent using many attributes.

In this work, we first investigate the methods to fully integrate KP-ABE encrypt, `EvalPK` and `EvalCT` operations in the `PALISADE` library while running all other KP-ABE operations in CPU. For this, we develop CUDA codes to run on GPU and introduce certain modifications to the library's API regarding typecasting, including the GPU kernels and modifying the `Cmake` scripts. As a result, our codes are compiled along with `PALISADE` seamlessly[1]. Also, even though the main library supports multi-precision integers, the built-in headers are not intended for usage in CUDA codes, which is an obstacle. Therefore, we have to include our GPU implementation for multi-precision (128-bits integers) arithmetic and incorporate it in the `PALISADE` library. Users and developers of `PALISADE Trapdoor`[2] libraries with `CUDA` installed on their machines are able to build and run our GPU accelerator without major extra builds.

The major execution bottleneck in KP-ABE operations is the multiplication of polynomials of a very high degree, whose coefficients are multi-precision integers. For instance, depending on the number of attributes, KP-ABE employs polynomials whose degrees are

[1]We provide our codes and other files for compilation and build at **www.github.com/RoronoaZ/KPABE-GPU.git**

[2]Another library built by PALISADE developers, which contains the KP-ABE scheme API.

as high as 2^{14} and coefficient sizes as large as 300 bit. We use a highly efficient GPU implementation of number theoretic transformation operation for polynomial multiplication. Our CUDA implementation of NTT is again seamlessly integrated with the PALISADE library and provides significant speedup values.

This work also introduces the first publish/subscribe communication application based on the KP-ABE scheme for providing superior security properties such as end-to-end encryption, ensuring the privacy of subscribers pertaining to the nature of their interest in the published messages. Our proposal provides true decoupling of publishers and subscribers and protects published messages against communication servers that receive, store, and forward messages. The proposed Pub/Sub system enjoys high performance provided by our GPU-base accelerator.

The paper is organized as follows: The section 2 will go through the works related to our topic. The following section 3 defines the necessary mathematical information and background to understand the cryptographic constructs in this scheme. Then, section 4 gives details about the KP-ABE scheme. Next, we present a concise explanation of the GPU-accelerated *Number Theoretic Transform (NTT)* implementation that we used in our work in Section 5. Later on, our proposed KP-ABE based publish/subscribe scheme is presented in Section 6. We discuss the acceleration of the scheme's primitives and the challenges faced during the development process in Section 7, while in Section 8, we present the experiments and the development environment's details. Finally, we conclude our work with Section 9 and one appendix B, including the full timings of our experiments.

2 Related Work

The publish/subscribe (Pub/Sub) communication system supports exchanging messages between information creators (producers) and consumers, who acquire such information by subscribing to a set of topics. The other party included in the Pub/Sub system is called the message intermediary (broker), which delivers the messages to the communication channel subscribers according to their submitted topics. This loosely-coupled communication protocol was first introduced in [19]. Although it ensures no direct communication between the publishers and subscribers, it is subject to data privacy breaches since, under specific scenarios, the messages are disclosed to the broker, which may act with malicious intentions or be subject to malicious intrusions. The other security concern is the security level of the encryption schemes used in the Pub/Sub protocols and their durability against quantum attacks.

Works in [22, 23] provide Pub/Sub implementations that address the security concerns by applying a proxy re-encryption scheme (*PRE*), which allows the broker or server to re-encrypt already encrypted data sent by the publishers so that the new ciphertext can be decrypted by interested subscribers. This method addresses the problem of the potential information disclosure at the broker side [22]. The authors in [23] propose two lattice-based *PRE* schemes that support key switching, allow complete decoupling between publishers and subscribers, and provide the ability to pass the access grant to other servers or brokers. The common aspect between the latter work and ours is that both use homomorphic encryption schemes. Furthermore, both works in [22, 23] and this paper address the confidentiality and privacy concerns in Pub/Sub communication system.

While they provide relatively fast solutions as re-encryption operation can be performed efficiently on off-the-shelf computers, PRE-based end-to-end secure Pub/Sub systems have several shortcomings as that in [22, 23]. First, the PRE scheme relies on a *policy authority*, which acts as an intermediary between publishers and subscribers and can see the content of the published messages as it generates public and secret keys for publishers. The access policies are formed very simply, and the policy authority is responsible for generating re-encryption keys for the proxy servers to perform the re-encryption operation. The access

policy simply determines who will receive a particular message, and the re-encryption must be repeated separately for every interested subscriber. If many subscribers are interested in the same message, this will be prohibitively expensive for a practical messaging application.

On the other hand, a Pub/Sub communication system based on KP-ABE provides more flexibility as much more involved policies can be expressed over a set of attributes that needs no policy authority. A communication server can homomorphically evaluate ciphertext without any re-encryption keys. Here, as homomorphic evaluation of ciphertext transforms an original ciphertext under an access policy, it is analogous to the re-encryption operation of a PRE-based solution. If a policy dictates multiple subscribers receive the same message, ciphertext evaluation is performed once for all subscribers interested in the message. In the PRE-based system, on the other hand, both the re-encryption key generation and re-encryption operations have to be repeated for every interested subscriber. In the KP-ABE-based solution, however, an authority referred to a private key generator in all identity and attribute-based encryption schemes generate secret keys per policy. Secret key generation is not particularly costly.

The authors in [22] state that the infeasibility of ABE schemes is due to the required expensive hardware for acceleration, which may not be available within reach of publishers for encrypting their data needs a re-examination; we intend to address this problem in this work. Here, we develop fast GPU implementations of KP-ABE encryption and evaluation functions built on top of the existing CPU implementation in [21] of the *KP-ABE* scheme. And we show that conventional GPU hardware, which is available even in many off-the-shelf desktop and notebook computers, can deliver an acceptable level of performance in the Pub/Sub systems, satisfying the need for computational power requirements. In addition, in Pub/Sub systems in which the same message is delivered to many subscribers, KP-ABE-based solutions offer computational advantages over PRE-based solutions.

In Table 1, we give a qualitative comparison of PRE- and KP-ABE-based end-to-end secure Pub/Sub communication services. Giving a fair quantitative comparison can be difficult as one may offer some computational advantages over the other depending on the use-case scenario. For instance, if one subscriber is interested in a published message and provided that all re-encryption keys are generated and given to the communication server acting as proxy encryption party, the re-encryption takes about 297 ms concerning the best PRE scheme that Polyakov et al. provided in [23], which gives an upper bound of 1 s for the generation of re-encryption keys. A ciphertext evaluation operation in [25] finishes in 590 ms when the number of attributes in the policy is $\ell = 8$ while generating a secret policy key takes 151 ms. Both implementations use the ring dimension of $n = 8192$ and run on Intel i7 CPU. Note that if more than one subscribers are interested in the same message, the policy authority and the communication server must repeat the generation of re-encryption keys and re-encryption operations for every subscriber in the PRE-based solution. In the KP-ABE-based solution, the communication server performs the ciphertext evaluation only once for all interested subscribers, provided they all want the same message using the same policy. As can be understood from these quantitative comparison attempts, care must be taken when interpreting the execution time results as they are highly dependent on the application scenario.

The implementation in this work introduces no fundamental changes in the construction of the key-policy attribute-based encryption (*KP-ABE*) scheme. It preserves the same hardness level as the Shortest Vector Problem (SVP), which is an NP-hard problem. The extension of the LWE problem to the RLWE construction [24] is mainly proposed to reduce memory overheads and increase the performance of the cryptographic operations that form the bottleneck in this scheme.

With the emergence of mathematical foundations for secure post-quantum cryptography, such as lattice-based, hash-based, and code-based cryptography, a framework for building secure applications has been formed since then. For instance, the work in [8] introduces

Table 1: A qualitative comparison of PRE-based and KP-ABE-bases Pub/Sub communication systems

	PRE-Based Pub/Sub	**KP-ABE-based Pub/Sub**
Third Party	Policy authority (PA)	Private key generator (PKG)
Access Policy	Subscribers inform policy authority	Attribute-based and dynamic
Ciphertext processing	Re-encryption	Homomorphic evaluation
Key for ciphertext processing	Re-encryption keys	No keys needed
Repetition of ciphertext processing	per user	per policy
Private key generation	Policy Authority for publishers	Private key generator per policy
End-to-end security	Policy Authority can decrypt all messages	No authority alone can decrypt messages
Privacy of subscribers	Access policies known by PA	Access policies can be hidden from PKG
Search over messages	Not possible	Attribute based search

a variant of the *attribute-based encryption*, called the *ciphertext policy attribute-based Encryption* (CP-ABE). It defines the access policy and links it with the ciphertext such that the policy cannot be changed after the encryption. On the other hand, the *KP-ABE* links the access policy with a set of attributes chosen in various means by subscribers wishing to decrypt a ciphertext. In this design, since the policy is not defined by the sender (encryptor), it can be decided after the encryption operation. Moreover, it can be a changing policy where different message receivers holding different attributes will be able to decrypt it after getting the policy keys from a private key generator.

We adopt the particular KP-ABE construction, which is is the lattice-based *KP-ABE* scheme in [16], implemented in the `PALISADE` library [21] and GPU-accelerated in [20]. Although the state-of-the-art implementation in [21] provides a fast encryption operation, the homomorphic encryption and evaluation of ciphertexts and public keys are the bottlenecks of this scheme due to its high number of vector-vector and matrix-vector multiplications. While another optimization by the work in [25] accelerates KP-ABE operations significantly, its GPU acceleration has not been reported in the literature. Therefore, no evaluation of the new technique has been reported for higher number of attributes.

In this paper, we provide the fast and memory-efficient GPU implementation of the fastest lattice-based KP-ABE scheme in [25]. We also propose a Pub/Sub communication system with higher flexibility of forming access policies and superior security properties and show that it is practical. The GPU implementation is fully integrated to `PALISADE` as an accelerator, therefore, all other KP-ABE operations, which needs no acceleration, can still be performed in the host CPU. Our solution can be used to accelerate other somewhat homomorphic encryption schemes implemented in `PALISADE`, such as BFV [26], BGV [27], and CKKS [28].

3 Preliminaries

We present the required mathematical background regarding the lattice-based cryptography and the *KP-ABE* scheme.

General Notations We employ bold letters to denote vectors and calligraphic letters for sets and similar mathematical structures such as rings. We use n to refer to the ring dimension for the polynomial ring $\mathcal{R} = \mathbb{Z}[x]/\Phi(x)$, where $\Phi(x) = x^n + 1$ is cyclotomic polynomial of degree a power of two. Arithmetic operations in KP-ABE usually occur in the ring $\mathcal{R}_q = \mathcal{R}/qR$, where the polynomial coefficients are integers modulo q, $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$, and considered in the range $(-\lfloor q/2 \rfloor, \lfloor q/2 \rfloor)$ unless stated, otherwise. A polynomial in

$A \in \mathcal{R}$ can also be viewed as a vector of its coefficients over $\mathbb{Z}$, namely $A \in \mathbb{Z}^n$. The modulo q reduction operation can be applied to a polynomial's coefficients, $[A]_q = A \bmod q$; then, $[A]_q \in \mathbb{Z}_q^n$.

For efficiency, we use RNS arithmetic [18] as we use the isomorphism $\mathcal{R}_q \approx \mathcal{R}_{q_0} \times \ldots \times \mathcal{R}_{q_{t-1}}$ with $q = \prod_{i=0}^{t-1} q_i$, where q_is are smaller primes, which usually fit in the word size of a computer. For a positive integer base b, a non-negative integer $a < b^k$ can be decomposed into digits $a_0, \ldots, a_{k-1}$, where $a_i < b$, such that $a = \sum_{i=0}^{k-1} a_i b^i$.

For a positive non-zero m, we define $\mathcal{R}_q^{m \times m}$, $\mathcal{R}_q^{1 \times m}$, $\mathcal{R}_q^m$ as a matrix, row vector, and a column vector of ring elements in $\mathcal{R}_q$ respectively. The value of m is determined based on the following formula from [20]:

$$m = \left\lceil \frac{\log_2(q_i)}{\log_2(b)} \times t \right\rceil + 2, \tag{1}$$

where we assume smaller primes q_i are of the same size.

The sampling from a discrete uniform random distribution is shown by the notation $a \leftarrow_U \mathbb{Z}_q$, $A \leftarrow_U \mathbb{Z}_q^n$, where A is, in fact, the element of $\mathcal{R}_q$. Also, $B \leftarrow D_{\mathcal{R},\sigma}$ stands for random sampling from the distribution $D_{\mathcal{R},\sigma}$ with zero mean and a small standard deviation of σ, where $B \in \mathcal{R}$.

Ring Learning with Errors problem The most important hard problem employed to secure the KP-ABE scheme is known as the ring learning with errors (RLWE) problem. Suppose s is an arbitrary polynomial in $\mathcal{R}$ and $a \leftarrow_U \mathcal{R}_q$ and an error polynomial $e \leftarrow \boldsymbol{D}_{R,\sigma}$. Given these, we define the hardness of two RLWE problems as follows:

1. RLWE Search Problem: given the pair of $(a, as + e)$, it is hard to find r.
2. RLWE Decision Problem: given the pair $(a, as + e)$ and b is randomly sampled in $\mathcal{R}_q$, it is hard to distinguish between $(as + e)$ and b.

The security level of the implementation is determined by the modulus q, the ring dimension n, and the standard deviation value σ in the distribution $D_{\mathcal{R},\sigma}$ using the inequality from [30]

$$n \geq \frac{\log_2(\frac{q}{\sigma})}{4 \log_2(\delta)}, \tag{2}$$

where δ is the Hermite factor, which is one of the determining factors for security level. For instance, $\delta = 1.006$ provides ~ 100 bits of security.

4 Key-Policy Attribute-Based Encryption - Kp-abe

The Key-Policy Attribute-Based Encryption scheme allows senders to perform the encryption on a set of attributes and define a compact-sized private (policy) key associated with an access policy to permit the correct decryption of specific ciphertexts only to receivers holding the correspondent policy key. Assume we have a set of attributes $\mathcal{X} = \{x_1, x_2, ..., x_\ell\}$ of size ℓ, which we use to determine access policies. Assume also that each attribute $x_i \in \{0, 1\}$ takes binary values indicating whether a required attribute exits or not. Attributes can pertain to receivers as well as messages.

We can define an access policy as a circuit of `NAND` gates over a set of attributes. For example, assuming $\ell = 8$ such that $\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$, where the access policy can be defined as

$$\begin{aligned} f(x_1, x_2, ..., x_8) = (x_1 \wedge x_3) \vee (x_2 \wedge x_4) \\ \vee (x_5 \wedge x_6) \vee (x_7 \wedge x_8). \end{aligned} \tag{3}$$

The access policy is expressed as a Boolean function or logic circuit in this example. As NAND gates are universal, any access policy that can be expressed as a logic function can be realized using only NAND gates. As homomorphic operations over ciphertext and public keys are defined as arithmetic operations, each NAND gate is written as $\neg(x \wedge y) = 1 - xy$. Then, the policy in Eq. 3 are written as

$$\begin{aligned} f(x_1, x_2, ..., x_8) = (1 - x_1x_3) \times (1 - x_2x_4) \\ \times (1 - x_5x_6) \times (1 - x_7x_8). \end{aligned} \tag{4}$$

The expression in Eq. 4 is referred to as the evaluation circuit for the policy in Eq. 3 as both ciphertext and the public keys corresponding to the attributes in the policy are homomorphically evaluated using this circuit. In our implementations in this work, we use benchmark evaluation circuits to assess the efficiency of the KP-ABE schemes. Figure 1 is an example of a benchmark evaluation circuit with $\ell = 16$ attributes. The actual policies can be defined in various ways, whose corresponding evaluation circuits are most likely not more complicated than our benchmark circuit with the same number of attributes.

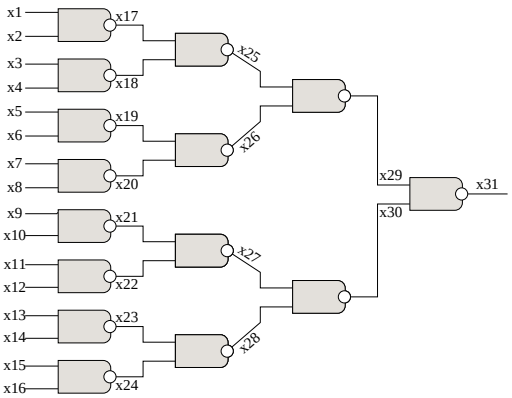

Figure 1: An illustration of a policy circuit of $l = 16$ and depth 4. Except from the input attributes, the rest of them where $l > 16$ are the attribute values on the circuit wires which form the inputs of gates at level i where $i \in [1{:}4]$ for this example.

The *KP-ABE* scheme consists of five main functions as follows:

1. $\texttt{Setup}(1^\lambda, \ell) \rightarrow \{\text{MPK, MSK}\}$: Given the number of attributes ℓ and the security parameter λ this phase generates a master public key (MPK) and master secret key (MSK), the latter of which is known to and used by a public key generator for generating policy secret key for ciphertexts that can be decrypted under the corresponding access policy. Note that MPK, used during encryption, includes row vectors of ring elements, namely $\boldsymbol{A}, \boldsymbol{B}_i \in \mathcal{R}_q^{1\times m}$ for $i = 0, 1, \ldots, \ell$ and $\beta \in \mathcal{R}_q$; namely $\text{MPK} = (\boldsymbol{A}, \boldsymbol{B}_0, \ldots, \boldsymbol{B}_\ell, \beta)$. We can think that one public key component $\boldsymbol{B}_i$ corresponds to the attribute x_i for $i = 1, \ldots, \ell$.

2. $\texttt{Encrypt}(M, \mathcal{X}, \text{MPK}) \rightarrow \boldsymbol{C}$: The sender encrypts the message M using the master public key and the set of attributes $\mathcal{X}$. For the ciphertext we can write $\boldsymbol{C} = (\boldsymbol{C}_{\text{in}}, c_1)$, where $\boldsymbol{C}_{\text{in}} \in \mathcal{R}_q^{1\times(\ell+2)m}$ and $c_1 \in \mathcal{R}_q$. Namely, $\boldsymbol{C}_{\text{in}} = (\boldsymbol{C}_A, \boldsymbol{C}_0, \ldots, \boldsymbol{C}_\ell)$, and $\boldsymbol{C}_A, \boldsymbol{C}_i \in \mathcal{R}_q^m$ for $i = 0, \ldots, \ell$.

3. $\texttt{Evaluate}(f, \boldsymbol{C}_A, \boldsymbol{C}_0, \mathcal{C}, \boldsymbol{B}_0, \mathcal{B}) \rightarrow \{\boldsymbol{C}_{policy}, \boldsymbol{B}_{policy}\}$: The evaluation function takes the policy circuit f and a subset of the public keys $\mathcal{B}$ and a subset of ciphertexts $\mathcal{C}$ corresponding to the attributes in f; i.e., $\mathcal{B} \subset \{\boldsymbol{B}_1, \ldots, \boldsymbol{B}_\ell\}$ and $\mathcal{C} \subset \{\boldsymbol{C}_1, \ldots, \boldsymbol{C}_\ell\}$.

It homomorphically evaluates the ciphertexts and the public keys pertaining to the access policy circuit f. As a result, it returns the evaluated public key and ciphertext $\boldsymbol{B}_{policy}, \boldsymbol{C}_{policy} \in \mathcal{R}_q^m$, respectively. If the access policy does not change (i.e., fixed policy case), $\boldsymbol{B}_{policy}$ can be obtained using a separate evaluation function $\texttt{EvaluatePK}(f, \boldsymbol{B}_0, \mathcal{B}) \rightarrow \boldsymbol{B}_{policy}$ once. Then, the ciphertext evaluation is performed for each ciphertext using $\texttt{EvaluateCT}(f, \boldsymbol{C}_A, \boldsymbol{C}_0, \mathcal{C}) \rightarrow \boldsymbol{C}_{policy}$.

4. $\texttt{KeyGen}(\boldsymbol{A}, \beta, \boldsymbol{B}_{policy}, \text{MSK}) \rightarrow \alpha_{policy}$: Given the master public key MPK, the master secret key MSK, and the policy public key $\boldsymbol{B}_{policy}$ for the policy f, the key generation function, executed by the private key generator, returns a secret policy key, α_{policy}, which is used to decrypt the ciphertext $\boldsymbol{C}_{policy}$. The policy secret key is sent in a secure way to the subscribers that satisfy the access policy.

5. $\texttt{Decrypt}(\boldsymbol{C}_{policy}, c_1, \boldsymbol{C}_A, \alpha_{policy}) \rightarrow \hat{M}$: Given the evaluated ciphertext C_{policy}, the other parts of the ciphertext, and the secret policy key the decryption results in $\hat{M}$ such that $M = \hat{M}$ when the secret policy key is correct.

Since we target the acceleration of the encryption (`Encrypt`) and the evaluation (`Evaluate`, `EvaluatePK`, `EvaluateCT`) functions, we briefly introduce their algorithms [3] as they show the operations that can benefit from the GPU acceleration. We also explain our approach to minimize their latencies or increase their throughput.

4.1 Encrypt

The GPU implementation of KP-ABE encryption can be made much faster than the CPU implementation as demonstrated in [20] with a GPU implementation. As it consists of a considerable number of involved arithmetic operations such as $\mathcal{R}_q$ multiplication, which can be carried out concurrently and take advantage of our highly optimized GPU implementation of the NTT operation, KP-ABE encryption, the steps of which are given in Algorithm 1, is a natural candidate for GPU acceleration. In particular, Step 3 and step 4 and the $\mathcal{R}_q$ multiplication, βs, in Step 5 of Algorithm 1 benefit from GPU acceleration.

Here, $\boldsymbol{G}$ is a primitive row vector of constant polynomials required for generating a $\boldsymbol{G}$-lattice extended by two 0 polynomials to match the row size m of polynomial vectors $\boldsymbol{A}$ and $\boldsymbol{B}_i$. The polynomial vectors $\boldsymbol{e}_0$ and $\boldsymbol{e}_A$ represent the noise or error component in the encryption added to the ciphertext vectors. The noise levels in the ciphertext as in the case of all lattice-based homomorphic encryption schemes, should not exceed a decryption threshold τ to guarantee correct decryption.

4.2 Evaluation

The homomorphic evaluation of public key vectors $\boldsymbol{B}_i$, ciphertext vectors $\boldsymbol{C}_A$, and $\boldsymbol{C}_i$ are performed using the functions `EvalPK` and `EvalCT`, which are combined into one function `Evaluate` in the `PALISADE` library, whose steps are detailed in Algorithm 2. The algorithm takes the public key vectors $\boldsymbol{B}_i$ and ciphertext vectors $\boldsymbol{C}_i$ in the policy circuit as inputs, performs the evaluation for each gate of the policy circuit, which is the benchmark circuit of the type in Figure 1, and returns the evaluated ciphertext and public key vectors $\boldsymbol{C}_{policy}$ and $\boldsymbol{B}_{policy}$, respectively. As a typical policy circuit does not contain all attributes, the public key and ciphertext vectors come from a smaller subset of all vectors, namely $\mathcal{B}$ and $\mathcal{C}$, and their cardinality can be written as $|\mathcal{B}| = |\mathcal{C}| = \ell'$. For simplicity, we designate the public key and ciphertext vectors in the policy circuit with indices starting from 0 to ℓ'.

During the homomorphic evaluation, the operation in the third step of Algorithm 2 produces a matrix of size $m \times m$, referred as the digit-decomposition matrix, $\boldsymbol{\Psi}_i$, which

[3] The algorithms referred to in this section can be reached in Appendix A

contains polynomials in $\mathcal{R}$, whose coefficients are smaller than the base b and distributed with a zero mean $\mu = 0$. The `NAFDECOMP` is a technique introduced first in [20] to limit noise growth using the base $b = 2$. The technique is later extended using higher base values in [25]. We adopt the latter approach as using higher bases, b, results in smaller m values, which minimizes the number of polynomials in public key and ciphertext vectors leading to significant computational advantage.

In the fourth and fifth steps of Algorithm 2, the homomorphic evaluation is performed on the ciphertext and the public key vectors using the `NAND` gate representation of the benchmark policy circuit in Eq. 4. The evaluation loops over the gates of the policy circuit, and the evaluated $\boldsymbol{C}_{policy}$ and $\boldsymbol{B}_{policy}$ are obtained in the output wires of the last gate. In Step 4 of the algorithm, a relatively large number of high-degree polynomials in $\boldsymbol{\Psi}_i$ and $\boldsymbol{B}_{2i}$ are multiplied using the NTT algorithm. Therefore, our GPU implementation of the NTT algorithm and other optimization techniques for GPU can significantly accelerate the KP-ABE homomorphic evaluation function in Algorithm 2. Our implementation of the algorithm is discussed further in Section 7.

4.3 Fixed Policy

The policy does not change often in certain application scenarios (e.g., publish/subscribe communication protocol). For instance, subscribers can communicate their access policy to the communication server, which computes the public key vectors $\boldsymbol{B}_i$, where $i \in [\ell' + 1, 2\ell' - 1]$ once using Algorithm 3 and stores them in memory. Furthermore, if the policy circuit is sufficiently simple and ℓ' is a small integer, the decomposition matrices $\boldsymbol{\Psi}_i$ for $i = 1, \ldots, \ell' - 1$ can be stored in the GPU memory, which can eliminate all NTT computations in the homomorphic evaluations of ciphertext vectors.

Algorithm 4 evaluates the ciphertext vectors in $\mathcal{C}$ in a fixed policy scenario, where the set of attributes and evaluated public keys are given as input. Consequently, the execution time of `EvalCT` function for the fixed policy will be less than when the policies are changing and thus, we have to run the `Evaluation` function. In the end, we will save $(\ell' \times \tau_{\texttt{EvalPK}} \times \tau_{transfer\_to\_gpu})$ seconds, where `EvalPK` and $\tau_{transfer\_to\_gpu}$ stand for the execution times for computing the public key vectors in the policy circuit and for time expended in transferring $\boldsymbol{B}_i$ for $i = 1, \ldots, 2\ell' - 1$, respectively.

4.4 Changing Policy

In the changing policy scenario, such as when the subscribers perform an attribute-based search over the encrypted messages published to the data space implemented by the communication server, the policy can possibly change with every query submitted by the subscribers. Then, the `Evaluation` function has to be executed, and the values of both C_is and B_i vectors are sent to and evaluated on the GPU kernel. In the changing policy scenario, m^2 NTT transformations at each gate in the policy circuit are performed.

5 Gpu-based Ntt Implementation

For NTT[4], we use the merge in-place method described in Algorithm 5, which is based on the factorization of the polynomial $x^n + 1$ into n one-degree polynomials. Therefore, we do not need post or pre-processing steps, and the output, $\boldsymbol{a}$, is a vector in bit-reversed order. Thus, it is unnecessary to re-order the output provided they are consistently kept in bit-reversed order while in the NTT domain. In Algorithm 5, we use a vector of n integers in $\mathbb{Z}_q$ both for the input polynomial and the output vector. Hence the algorithm is in-place.

[4]The algorithms referred to in this section can be reached in Appendix A

On the other hand, designing an NTT algorithm suitable for GPU implementation is more involved as a straightforward implementation of Algorithm 5 will likely be very inefficient. Algorithm 5 has a recursive nature and works on independent NTT instances of smaller sizes as the outer loop iterates. For instance, when $n = 4096$ while the first iteration computes one 4096-point NTT, two independent 2048-point NTT operations are computed in the second iteration. The number of NTT computations doubles with every iteration while each size halves. In the butterfly operation (see Steps 8-11 of Algorithm 5), which is the main operation of NTT computations, one GPU thread reads from two memory locations and writes into the exact locations. Therefore, when the number of threads in a GPU block, tc, is more than or equal to the half NTT size, i.e., $tc \geq n/2$, one GPU block can compute $2tc$-point NTT efficiently. Since single block threads use the same fast shared memory in GPU, NTT can be computed in one kernel. The method, given in Algorithm 6, is called NTT Kernel 2.

When $tc < 2n$, on the other hand, the method in Algorithm 6 becomes inefficient as the threads from different blocks should synchronize, which requires accessing global memory. Therefore, a new algorithm that minimizes global memory access is essential for efficient implementation. We adopt the approach in [31] and use Algorithm 7 for the first couple of NTT iterations until the sizes of NTT operations get sufficiently small so that they can be computed using Algorithm 6. For instance, a typical block size in contemporary GPUs is $tc = 1024$, which can compute up to 2048-point NTT without accessing global memory for synchronization. If we compute an 8192-point NTT, its first two iterations are computed using Algorithm 7 and the remaining 11 iterations with Algorithm 6.

There is a significant difference in our approach to performing multiplication in $\mathcal{R}_q$ when compared to one in [20], in which a carrier prime of goldilocks form ($P = 2^{64} - 2^{32} + 1$) is used for NTT operations. While the Goldilocks primes offer certain advantages for fast NTT arithmetic, our approach using arbitrary primes for RNS arithmetic has two advantages over the one in [20].

First, we use fewer word-sized primes while the method in [20] has to work with relatively more minor primes. For example, we have the following constraint for Goldilocks prime P: $P > n \cdot q_i^2$, where the primes q_i are used for RNS arithmetic. For 256 attributes, where $n = 2^{12}$ and $\log_2 q = 150$, one can only use 25-bit primes, q_i for RNS arithmetic, which will result in using at least six such primes to attain 150-bit modulus, q, for the ring $\mathcal{R}_q$. On the other hand, our approach can use 64-bit primes and therefore needs only three primes. Second, as we use the primes that are the factors of the modulus q, we have the isomorphism $\mathcal{R}_q \approx \mathcal{R}_{q_1} \times \ldots \times \mathcal{R}_{q_t}$, where q_i are word-sized primes employed in RNS arithmetic. In the method in [20], which uses the carrier modulus, this isomorphism is lost, necessitating an inverse NTT operation after every ring multiplication in [20]. On the other hand, this is not necessary in our approach due to the isomorphism.

6 Proposed Kp-abe based Publish/Subscribe Scheme

The Pub/Sub communication model is a framework of asynchronously exchanging messages between publishers, which send messages, and subscribers, which consume published messages, selected based on their preferences. A communication server, the Proxy, facilitates storing and forwarding published messages to interested parties. A subscriber can base his/her preferences for messages to receive on different attributes of the messages, such as message type and classification and even the words in the published messages. Publishers are entirely decoupled from subscribers as the former do not know who is accessing messages and subscribers' preferences. Nevertheless, the Proxy usually needs to know all those details and contents of messages to deliver them to interested subscribers in the existing implementation of Pub/Sub systems. Therefore, classical publish/subscribe communication systems suffer from the fact that neither end-to-end security nor subscribers'

privacy is ensured. Our proposed Pub/Sub system provides additional security and privacy guarantees formulated in the following definitions.

Definition 1 (End-to-end security in Pub/Sub system)**.** Pub/Sub system provides end-to-end security if a message is revealed only to subscribers interested in and has access rights to the message.

Definition 2 (Subscriber Privacy)**.** The Pub/Sub system supports subscribers' privacy if the messages subscribers receive are not revealed to other parties, including publishers of messages and the communication server (Proxy).

At the end of the section, we show that the proposed Pub/Sub system provides both. The proposed system can take several forms depending on the application scenario and the context. An example Pub/Sub scenario depicted in Figure 2 allows publishers to encrypt messages using the set of all possible attributes and subscribers to access messages based on access policies, for which a formal definition is given in the following.

Definition 3 (Access policy (f))**.** The access policy for a published message is a Boolean function defined over a subset of attributes, which determines subscribers that can decrypt the message.

Attributes can pertain either to messages or to subscribers. In the former case, the attributes are about messages, such as their subjects, themes, or the words contained in messages. Then, by categorizing them, subscribers are more likely to define the access policy to show their interest in receiving a certain subset of published messages. For instance, they may be interested in messages containing certain keywords or labeled in certain subjects. In this case, as only subscribers know the access policy, encrypted messages must be homomorphically evaluated after publication, which is possible only with the KP-ABE scheme.

In the latter case, the attributes are about the subscribers, such as their access rights or roles, which give them specific access rights. A typical example is role-based access control systems [32,33], in which a central authority usually determines the access rights of roles. Publishers may or may not know who will access their messages during publishing, i.e., the access policy. If the publisher knows the access policy, it can use the ciphertext policy attribute-based encryption scheme (CP-ABE), for which various efficient proposals are in the literature [8–12]. Nonetheless, if publishers do not know the access policy for a particular message, KP-ABE is the only alternative to support end-to-end security in the most strict sense.

In the scenario given in Figure 2, the publisher forwards the homomorphically encrypted message $\boldsymbol{C}$ to the proxy server over a secure channel, which can happen any time independent of the following operations, which needs to be performed in the order explained:

1. The access policy f, which N subscribers share, is communicated to the Proxy and the private key generator (PKG) over a secure channel. A list of access policies is kept at the Proxy and the PKG.

2. The Proxy performs the `EvalPK` function to generate the policy public key $\boldsymbol{B}_{policy}$ and sends it to the PKG.

3. The PKG performs the `KeyGen` function to generate the secret key for the policy f, sk_{policy} using the master secret key MSK. It then checks the subscribers' list registered to f. Finally, if the access rights of a particular subscriber allow it to exercise f on any message published, the secret policy key sk_{policy} is delivered to it over a secure channel.

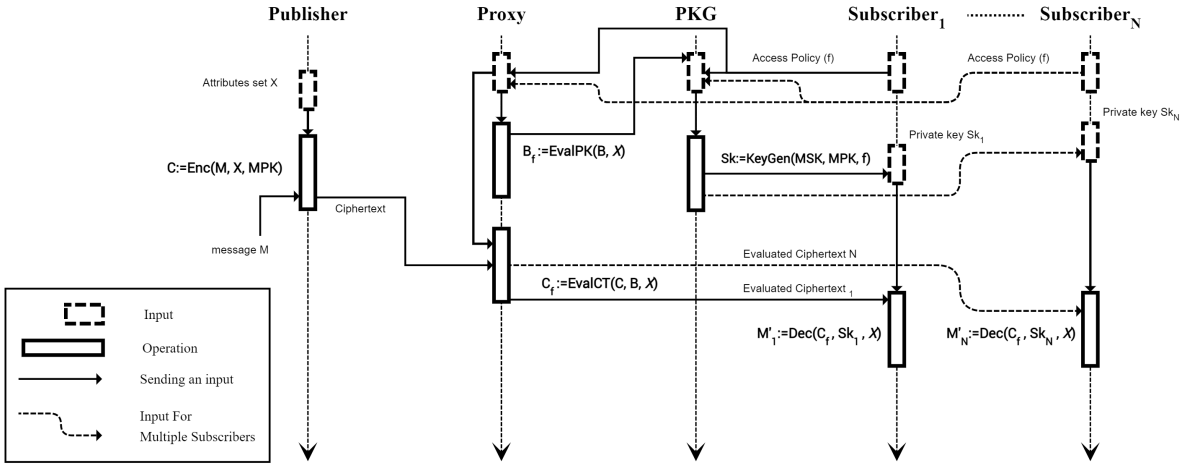

Figure 2: The general architecture of the publish/subscribe scenario that this work tackles.

4. When ciphertext messages arrive, the proxy checks whether their attributes satisfy any access policy in the list, then it homomorphically evaluates the message for matching access policies (`EvalCT`). Finally, the Proxy transmits, over a secure channel, the resulting ciphertext $\boldsymbol{C}_{policy}$ to the subscribers registered in f.

In order to prove that the proposed Pub/Sub system fulfills the security and privacy guarantees formulated in Definition 1 and Definition 2, we need the following assumptions:

Assumption 1 (Secure Channel)**.** *The parties have access to pair-wise secure channels, which provide confidentiality, authentication, and integrity for the messages exchanged.*

The secure channel assumption can be satisfied by employing the standard transport layer security protocol, which secures more than 90% of the Web traffic.

Assumption 2 (Non-collusion)**.** *The PKG and the Proxy do not collaborate to access the content of messages published.*

Naturally, if the ciphertext message posted to the Proxy $\boldsymbol{C}$ or any evaluated ciphertext message $\boldsymbol{C}_{policy}$ the PKG captures, it can decrypt it as it holds the master secret key.

Assumption 3 (Hidden Attributes)**.** *The semantics of attributes are not revealed to the Proxy, which knows only the pseudonyms of the attributes.*

The attributes are treated as binary values, such as their existence or non-existence in a particular message, so they do not need to be revealed to the Proxy. Nonetheless, the Proxy can perform frequency analysis and learn about the frequently used attributes.

Given the definitions and assumptions, we can now prove that the proposed Pub/Sub-messaging system provides the following security and privacy properties.

Theorem 1. *The proposed Pub/Sub messaging system provides* **end-to-end security** *for the exchanged messages.*

Proof. Due to Assumption 1, messages are encrypted twice when they are in public channels, once with the KP-ABE scheme and the second with the classic encryption scheme provided by the secure channel. Herefore, no parties can decrypt and access their contents, including PKG, if they are not authorized. In the Proxy, however, the messages

are encrypted only with the KP-ABE scheme. Knowing neither the master secret key nor the secret policy key, the Proxy cannot decrypt the messages. Assumption 2 stipulates that the Proxy does not share the messages with the PKG, which cannot decrypt them. □

Theorem 2. *The proposed Pub/Sub messaging system supports subscribers' privacy by preventing their access policies from being exposed.*

Proof. By Assumption 1, when a subscriber sends its access policy to the Proxy and PKG via a secure channel, no party, including the subscribers, can see the access policies other than the communicating parties of the secure channel. By Assumption 3, the Proxy extracts partial information on the access policies while the PKG must know them as PKG is the party that determines and exercise the access rights in general. In summary, subscribers' privacy is protected fully against publishers and partially against the Proxy. □

The `PALISADE` library provides an example code for the *KP-ABE* scheme, and the implementation is aligned with Pub/Sub application scenarios outlined in this section. Therefore, we use it as a starting point to design our GPU-based Pub/Sub communication scheme implementation. Our implementation of the Pub/Sub application permits encrypting messages on GPU, using a large number of attributes (up to 128 attributes) by the sender. The receiver(s) (subscriber) only need to decrypt the messages associated with the same or a smaller set of attributes that matches the receiver's chosen attributes (defined previously in f). The evaluation process (also on GPU) can either be performed by the receiver or a third party. However, performing it by a proxy with sufficient computation power is preferred to avoid extra latency values and heavy workloads imposed by the relatively expensive homomorphic computations on the receiver side.

7 Acceleration of Kp-abe Encrypt and Evaluate Functions

The GPU or hardware acceleration of the encryption and evaluation functions of the *KP-ABE* scheme, which is implemented in the well-known library `PALISADE`, can be highly challenging due to many technical details. Even though `PALISADE` has official releases, it is still a library under development and comprises obstacles such as type casting functions and integration of GPU scripts within the main library (compiling scripts). Moreover, to make use of the *KP-ABE* main functions, it is required to install the separate `Trapdoor` library (also developed by [21]) and create/bring modifications to their compiling scripts.

7.1 Technical Challenges

In this section, we highlight the main issues encountered during the integration and development of our GPU implementation into `PALISADE` and how they are resolved in our implementation.

1. The integration of the `Trapdoor` API and the `PALISADE` library with `CUDA` code we developed is technically challenging as the task contains many minor implementation details. We provide a compiling script that enables the usage of both `PALISADE` and `Trapdoor` libraries alongside the integration of `CUDA` scripts. Our solution permits running `PALISADE` API (with our implemented functions) on both CPU and GPU.

2. It is involved and time-consuming to convert the elements of the polynomial rings $\mathcal{R}$ and $\mathcal{R}_q$ stored in highly complex and structured `DCRTPoly` (Double CRT)[5] data structures into more primitive data structures (arrays) acceptable by `CUDA` kernels. Then, we introduce data type casting functions to the existing APIs inside the main

[5]The Double CRT data type is used in `PALISADE`'s API to store both the polynomial and their coefficients in CRT form, which are optimized for memory access in CPU.

PALISADE library so that it is installed with its scripts once a rebuild is performed. For instance, for most polynomials, a matrix of DCRTPoly data structure is used in PALISADE. Each DCRTPoly instance encompasses k polynomials (under the *CRT* configurations) where k is the number of the smaller prime moduli employed in RNS arithmetic. The introduced type-casting functions will copy these polynomials into primitive arrays declared in the device's memory by preserving their order in the DCRTPoly matrix structure. The type-casting operation is reversed by copying back the polynomials from CUDA arrays into the DCRTPoly polynomials after the kernels' execution is performed since the resulting polynomials can be processed in the subsequent operations of the KP-ABE scheme in the CPU; Decryption for instance.

3. Performing the NTT operations (Forward & Inverse) is prohibitively expensive, especially in the homomorphic evaluation process, which incorporates overly many NTT operations. Therefore, we integrate a GPU implementation of the NTT operations discussed in Section 5 into our GPU-accelerated scheme.

7.2 Memory restrictions

The complexity of prohibitively resource-consuming computations imposed by the homomorphic evaluation operations (e.g., number of homomorphic multiplications and additions), plus the necessity of a considerable amount of memory to store the polynomials in the device memory, forms an obstacle to achieving efficient GPU implementation that can fully function even for a relatively high number of attributes. Therefore, we pursue two approaches (mentioned in Section 4), referred to as *Fixed Policy* and *Changing Policy*, to take advantage of different use case scenarios to interplay computation complexity and memory requirements. In the former case, where the policy is known beforehand and does not change often, we pre-compute and store the polynomial vectors $\boldsymbol{B}_{\ell'+i}$ for $i = 1, \ldots$ using the EvalPK function to avoid computing them every time for each ciphertext, which will be homomorphically evaluated for the same policy. This eliminates Step 4 in Algorithm 2 in a GPU kernel, which frees a substantial amount of GPU's main memory space, which can now be used for computing $\boldsymbol{C}_{\ell'+i}$ during the evaluation of the ciphertexts. In the latter case, where the policy changes, we must compute the polynomial vectors $\boldsymbol{B}_{\ell'+i}$ in run time. The latter case is much more challenging for given GPU resources in terms of memory requirements and computation complexity, and as we will show in the subsequent sections, there will be limits on the number of attributes in the policy if we want to keep the computational latency within reasonable ranges.

In our implementation, we use *Unified Memory* architecture while allocating memory for the polynomials, which creates a shared memory space between the host (CPU) and the device (GPU) such that when the device memory is depleted, the host memory will be used. Although this memory architecture does not alleviate the overhead of data transfers between the host and the device, it allows a larger memory space without manual intervention. Thus, it becomes possible to run the implementation with a much higher number of attributes.

As for the GPU implementation of the encryption function, there is no primary concern regarding the memory requirement since there are fewer polynomials and relatively less computation compared to the Evaluate function.

Regarding the Evaluate operation, our new API functions allow copying data types of PALISADE into arrays in GPU, where four large arrays dominate the GPU memory during the homomorphic evaluation operation. The set $\mathcal{C}$, which contains ciphertext polynomial vectors used in the policy, is the input to Algorithm 2. During the homomorphic computation, we generate $\boldsymbol{C}_{\ell'+i}$ for $i = 1, \ldots$ representing ciphertext vectors for the inner gates of the policy circuit. The significant memory consumption is due to the matrix

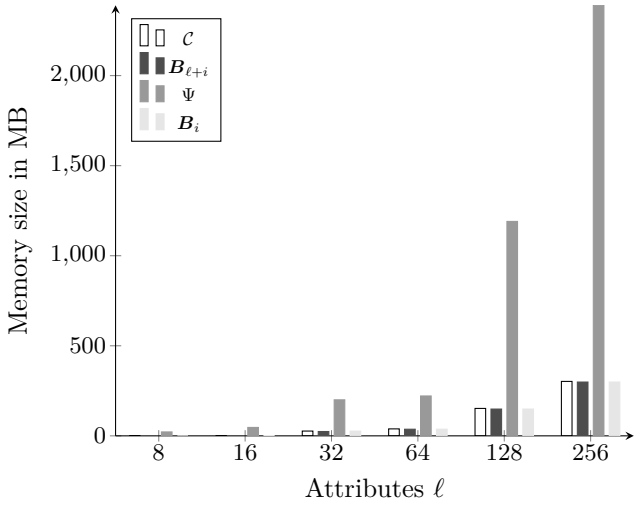

Figure 3: This plot shows the approximate amount of memory in **Megabytes (MB)** required to accomplish the evaluation process as the attribute number increases. The matrix of polynomials Ψ dominates the memory usage. This illustration matches the *Changing Policy* scenario, while the memory requirements decrease when applying the *Fixed Policy*

Ψ since it is a $\mathcal{R}_2^{m\times m}$ Double-CRT matrix of polynomials in `PALISADE`. Finally, there is the resulting ciphertext vectors $\boldsymbol{C}_f$ or $\boldsymbol{C}_{policy}$. Fig. 3 shows the increase in memory requirements as the number of attributes increases. It should be pointed out that the number of attributes, the ring dimension n, and the modulus size q considerably affect the amount of required memory for the evaluation process.

The vectors shown in Figure 3 are the main data structures that occupy the device memory. For instance, the polynomial vectors $\boldsymbol{B}_i$ and $\boldsymbol{B}_{\ell'+i}$ are also stored in the device memory. Herefore, for some CPUs and GPUs with modest main memory sizes, the number of attributes is limited to $\ell = 16$ or $\ell = 32$.

This work implements the *Fixed* and *Changing* policies. In the fixed access policy, we work on the assumption that the polynomial vectors $\boldsymbol{B}_i$, where $i \in \{\ell' + 1, 2\ell' - 1\}$, are pre-computed and stored on the device memory. Differently from [20], we copy at one go the required polynomial vectors $\boldsymbol{B}_i$ and $\boldsymbol{C}_i$ to a unified memory space to benefit from the high GPU memory bandwidth rates used in this work. Therefore, the overhead of data transfers is kept to a minimum. Furthermore, when the applied scenario is the changing policy, the NTT transformations alongside the evaluation of the public key vectors $\boldsymbol{B}_i$ and ciphertext vectors C_i are performed in one function `Evaluate`, as stated in Subsection 4.4.

7.3 Accelerating the encryption and evaluation operations

We develop two `CUDA` kernels for executing KP-ABE encryption and homomorphic evaluation operations in GPU, which are written in C language by adding the directives "`__global__`" or "`__device__`". Kernels preceded with `__device__` cannot be called from the host code, for which `__global__` before calling the CUDA kernel is used. It allows the compiler to identify it as a piece of code to execute on the GPU. GPU kernels come with a set of arguments and parameters, including the number of threads, the number of thread blocks, grid size, and the size of shared memory (in case the implementation uses shared memory among thread blocks). Additionally, `CUDA` streams are essential to run concurrent `CUDA` operations in case there are other streams than the default one to pipeline data transfers and computations.

```
// kernel.cpp
#include<...>
__global__ void Encrypt_Ker(uint128t * ctCin_d, uint128t * s_d, uint128t * g_d,
    uint128t * eA, uint128t * eCin, ...)();

Encrypt_Ker<<<ROWSIZE / BLOCKSIZE, BLOCKSIZE, shared_mem_size, EvalStream>>>(ctCin_d,
    s_d, g_d, eA, eCin, ...);
CUDA_SAFE_CALL(cudaEventSynchronize(stop));
```

The code snippet above shows a small code segment that calls a defined kernel with the selected dimensions (grid and block dimensions, threads per block in the first two arguments after the symbol `<<`. The next argument indicates the amount of the shared memory that will be used, and the last one stands for the `CUDA` stream allocated for the specific kernel.

The design in our implementation dynamically allocates the number of grids, blocks, and threads depending on the values of n, m, and the number of RNS primes[6], t. We use $(\frac{m \times n \times t}{MAX\_THR\_PER\_BLK})$ thread blocks, where $MAX\_THR\_PER\_BLK = 1024$ is the maximum number of threads per block for the GPUs used in our implementation.

Furthermore, we create `CUDA` streams to concurrently handle data transfers and computations. The dimension of each block is a multiple of the warp size (32) to increase the average utilization of thread resources. The transferring of polynomial coefficients from `PALISADE`'s data structures into GPU arrays is accelerated using `OpenMP`[7]. The transfer times are kept to minimum (less than 40 ms) for the largest possible parameters (ℓ, m, n, q).

Table 2: The hardware specifications

	Machine 1	**Machine 2**	**Machine 3**
CPU - Intel(R)	i9-7900X	Xeon(R) Gold 6152	i9-11900K
GPU	GTX 1080	Quadro GV100	RTX 3070 Ti
Memory (host)	32 GB	1 TB	32 GB
Memory (GPU)	8.5 GB	32 GB	8 GB
Bandwidth (GPU)	320.3 GB/s	868.4 GB/s	608.3 GB/s
Base clock (GPU)	1607 MHz	1132 MHz	1575 MHz
CUDA cores	2560	5120	6144

8 Experiments & Implementation Results

We compare our work with the state-of-the-art GPU and CPU implementations, namely [20] and [25], respectively. We use a machine with an Intel i9-7900X processor running at 3.30 GHZ with an `Nvidia GeForce GTX 1080` GPU with the *Pascal* architecture. Other GPUs, also used for comparison, are `Nvidia Quadro GV100` and `RTX 3070 Ti` with the *Volta* and *Ampere* architectures, respectively (see Table 2 for technical details). The operating system is `Ubuntu 20.04 LTS` with the compiler `g++ 9.4.0` and the CUDA toolkit `CUDA 11.4`.

We should note that we compare our implementation with the `PALISADE` library, which does not include a GPU implementation. However, we compare it with the GPU timings reported in work by Dai et al. [20], which also uses the `PALISADE` library and slightly different, but comparable GPU devices (`Titan X` and `Titan XP`). As mentioned by the authors of [20], their timings are estimated starting from a relatively low number of

[6]The implementation in `PALISADE` uses RNS arithmetic for large q sizes. It generates t smaller RNS primes to speed up the polynomial multiplications of a higher number of coefficients. In addition, it helps in the parallelization of coefficients multiplication.

[7]OpenMP is a multi-platform multithreading library that allows the use of threads to accelerate the programs' sequential execution times.

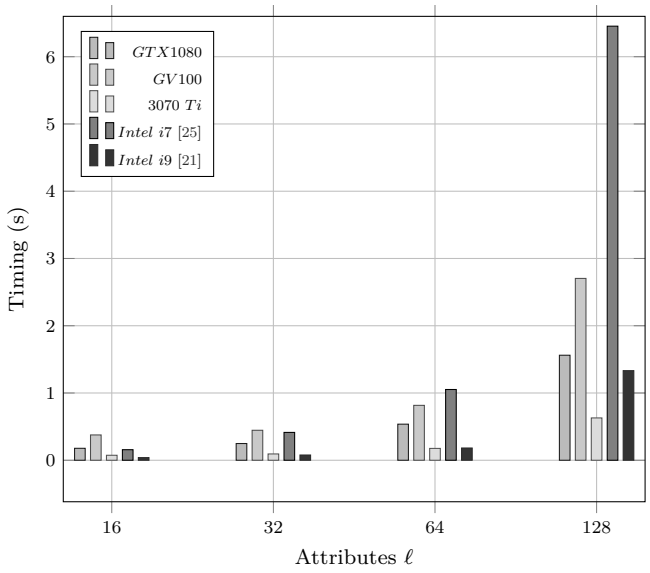

Figure 4: This plot compares the KP-ABE Encryption operation between the CPU runs and our GPU implementation on 3 different GPUs. The ℓ value starts from 16 until 128 attributes. These timings include the data copying between the CPU & GPU memories. The actual timing values are included in Appendix B.

attributes, i.e., $\ell > 32$, since their implementation is memory-restricted. On the other hand, our timings are accurate as they are the results of the actual runs of the GPU-accelerated KP-ABE scheme of `PALISADE`. We achieve successful homomorphic encryption and evaluation for 128 attributes with execution times less than 0.5 and 2.5 seconds, respectively (see Tables. 4–6 and 7). In the table, the encrypt kernel timing includes the sampling of the secret and the error polynomials (Steps 1, 2 and 3 in Algorithm 1), which runs in the host device while the homomorphic evaluation includes both `EvalCT` and `EvalPK` operations for Changing Policy scenario, but only `EvalCT` for Fixed Policy scenario.

8.1 Comparison with the CPU implementations

Figures 4 and 5 illustrate the comparison of execution times of our GPU-based and the CPU-based KP-ABE encryption operations, where the timings for a lower number of attributes are not included (for the actual timing values, see Table 7 in Appendix B). The difference between Figures 4 and 5 is that the second does not include the entire timing of the CPU-GPU data copying overhead, specifically, the `PALISADE`-GPU in-between communication. In Figure 5, the timing results labeled with *Intel i7* are taken directly from [25]. We also run KP-ABE encryption of `PALISADE` on a faster CPU, where the execution times are labeled as *Intel i9*. When measuring the execution times, we run KP-ABE encryption and KP-ABE evaluation functions with the same number of attributes and check if the ciphertext decrypts correctly; this is repeated several times, and averages are taken. Our GPU implementation on RTX 3070 Ti is superior to the *Intel i7* runs for all attribute counts. Moreover, our implementation performs better than the *Intel i9* processor runs for all attributes on RTX 3070 Ti. We achieve up to 2.7× speedups compared to the best CPU run for $\ell = 128$ (see Table 7 in Appendix B).

For homomorphic evaluation operation, Figures 6 and 7 show the execution times of the homomorphic evaluation on the selected GPU devices and Intel processors. Compared to the *Intel i9* processor runtimes, our GPU implementation is a minimum of 1.9 times

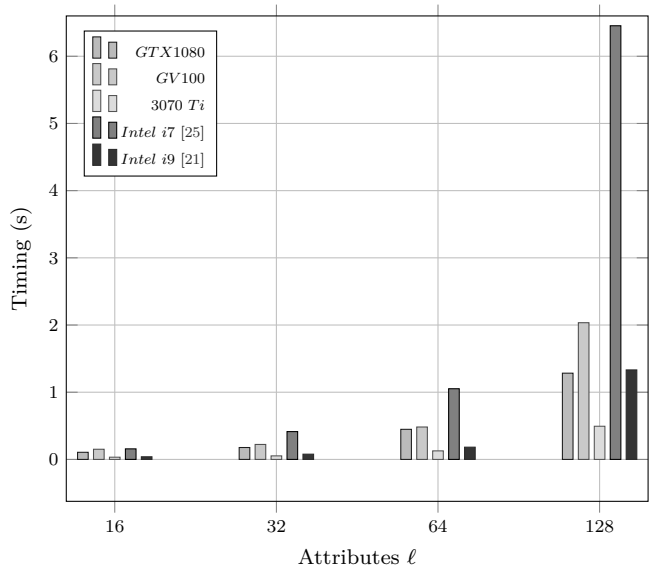

Figure 5: This plot compares the KP-ABE Encryption operation between the CPU runs and our GPU implementation on 3 different GPUs. The ℓ value starts from 16 until 128 attributes. These timings **do not** include the data copying between the CPU & GPU memories. The actual timing values are included in Appendix B.

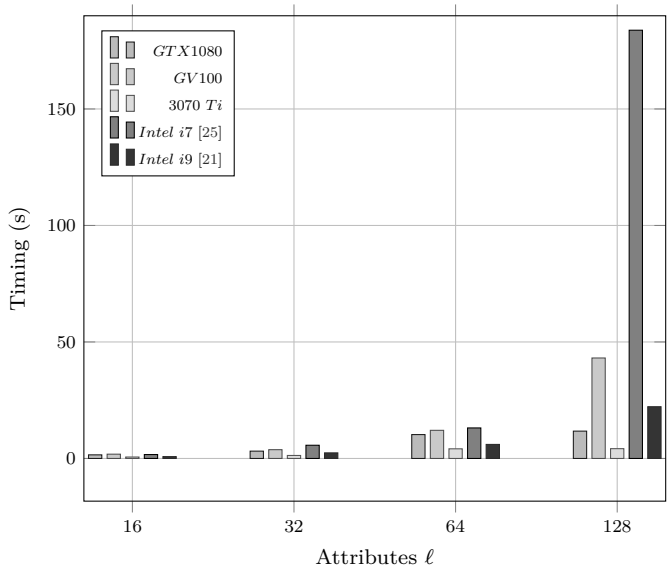

Figure 6: Comparison of the KP-ABE Evaluation operation between the CPU runs on the *Intel i*9 and *Intel i*7 [25] processors and our GPU implementation on 3 different GPUs. The timings include the CPU-GPU data copying overhead.

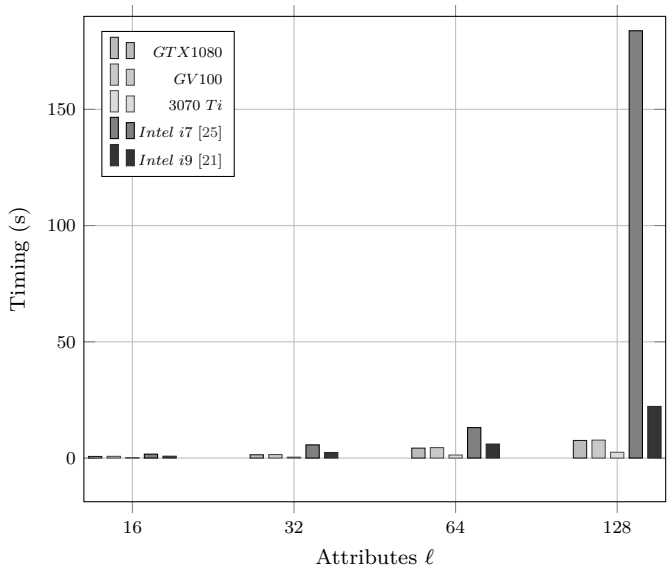

Figure 7: Comparison of the KP-ABE Evaluation operation between the CPU runs on the *Intel i*9 and *Intel i*7 [25] processors and our GPU implementation on 3 different GPUs. The timings **do not** include the CPU-GPU data copying overhead.

(for 2 attributes on the GV100) and a maximum of 22.3 times (for 128 attributes on the RTX 3070 Ti) faster (for detailed timing results see Table 4 in Appendix B). Furthermore, the acceleration ratio remains between 4 × and 12 × for the rest of the attributes. On the other hand, the ratios are at least 1.9 × (for GV100) and, at most, 40 × (for RTX 3070 Ti) compared to the *Intel i*7 processor. Finally, from Figures 7 and 6, it is observable that we have at least two GPUs (`GTX 1080` & `RTX 3070 Ti`) in which our GPU-based implementation is faster than the CPU-based KP-ABE implementation on the *Intel 9* processor.

8.2 Comparison with the GPU implementation

The GPU implementation presented in this work differs from that of Dai et al. [20] in that we fully integrated the GPU kernels within `PALISADE`'s API. This allows us to call our GPU kernels in lieu of generally slower CPU-based encryption and evaluation functions. On the other hand, Dai et al. use a standalone GPU implementation of the KP-ABE scheme, which is not integrated into PALISADE, and their implementation is not public. Moreover, their implementation used smaller parameters, including a smaller base b, compared to the parameters that can be used on `PALISADE` for KP-ABE.

Notably, the work in [20] estimates the timings of the homomorphic evaluation for the number attributes $l \geq 32$. Supposing that the previous GPU implementation uses unified memory architecture, those estimations may turn out to be optimistically low when working with larger attribute counts, modulus, and ring dimensions. The possible latency increase could be attributed to two potential factors. The first factor plays out when the GPU's memory is full, whereby extra CPU RAM memory will be allocated, leading to data transfers between host and device memories. The second factor affects the performance when the CPU RAM is also at its limit, and a SWAP partition is allocated on the hard disk drive. Then, the latency figures would significantly increase as well.

Therefore, a comparison of our work with the work of Dai et al. [20] is involved. Different GPU architectures employed in [20] add to the difficulty of achieving a fair comparison. To this end, we run our NTT and inverse NTT implementation on Titan X, on which the

timing of NTT-based polynomial multiplication is reported in [20] and include the results in Table 3. As can be observed from the table, the fastest GPU (RTX 3070 Ti) in our implementation runs the NTT and inverse NTT operations faster than Titan X. In the next section, we compare our NTT-based polynomial multiplication and the one reported in [20]

Table 3: Forward and Inverse NTT Timings (In Microseconds) on Titan X / RTX 3070 Ti GPUs

		64-bit 4096-NTT	
Ring size	NTT_Count	**Forward NTT**	**Inverse NTT**
2^{12}	4	16.58 / 16.61	6.27 / 6.22
	16	31.49 / 17.2	13.0 / 7.68
	32	50.26 / 25.77	23.36 / 13.59
	64	77.46 / 38.28	30.54 / 18.72
	128	130.3 / 76.24	58.6 / 33.85
2^{13}	4	28.92 / 18.48	11.25 / 7.48
	16	55.44 / 26.93	26.07 / 14.01
	32	87.76 / 38.82	36.7 / 21.0
	64	134.2 / 75.1	61.88 / 35.05
	128	264.6 / 136.07	116.41 / 62.24
2^{14}	4	48.49 / 23.45	21.24 / 10.03
	16	109.38 / 41.17	41.35 / 21.76
	32	187.31 / 80.33	80.8 / 36.9
	64	268.88 / 139.68	133.95 / 67.6
	128	542.13 / 259.18	250.78 / 124.26

8.2.1 Comparison of the polynomial multiplication

Dai et al. [20] approximate the polynomial multiplication as 2× Forward NTTs and 1× Inverse NTT. As such, they estimate the overall timing of $(256 \times 128 \times 4)$ multiplications and compute the average single multiplication latency. They use the ring dimension of 2048, and RNS primes less than 24-bit for coefficients (recall a carrier Goldilocks prime of 64-bit is used in their work). They report that one polynomial multiplication takes 0.94μs on Titan X.

Here, we developed a program to measure the polynomial multiplication using the NTT implementation of [29], following a similar approach as [20] to ensure a fair comparison. However, we set the ring size to 4096 for 64−bit coefficients. Therefore, since the ring sizes differ, we estimate our multiplication latency by dividing the timing by 2.18 (knowing that the algorithm's complexity is $n \log_2 n$). Furthermore, we use 60−bit RNS primes as opposed to the 24-bit primes used in [20]. This means that the implementation in [20] needs to perform $60/24 = 2.5$ more RNS primes than ours to reach the same modulus q. As timings are measured for batch execution of polynomial multiplication, this gives our implementation a 2.5× advantage in timings.

In Table 3, for 128 parallel polynomials multiplications for the ring dimension 4096, the time taken for one `Forward NTT` operation is $130.3\mu s$ while the timing for the `Inverse NTT` (INTT) is $58.6\mu s$. As we consider a polynomial multiplication as 2 NTTs and 1 INTT operation, an average latency of a polynomial multiplication is calculated as $\frac{130.3 \cdot 2 + 58.6}{128} \approx 2.49\ \mu s$. If we factor in 2.5× advantage due to our larger RNS primes, we can conclude that our implementation is estimated to be $\frac{0.94 \cdot 2.5 \cdot 2.18}{2.49} \approx 2$ times faster than the timing reported in [20].

8.3 GPU-based Publish/Subscribe Evaluation

Table 8 shows the timings of both homomorphic operations after running them with different parameters. Each row on Table 8 can be read as follows: in the 8^{th} row, the encryption includes 32 attributes while the access policy incorporates only 4 attributes for the evaluation operation. The ciphertext modulus size is 100 bits, and the ring size is 8192, with a base value equal to 2^{20}. Under the `Encrypt / Evaluation` column, the figures show the run times of encryption and evaluation for each GPU separated by a slash.

It is clear from the timings mentioned in the table that adopting such a communication protocol based on a lattice-based homomorphic scheme KP-ABE is no longer impracticable. Furthermore, the proxy with an off-the-shelf GPU can handle many homomorphic evaluation operations. Meanwhile, subscribers only decrypt their respective messages, which are relatively inexpensive operation compared to homomorphic encryption and evaluation. However, if the designed communication protocol lets the subscribers perform the evaluation locally, it requires only a GPU with similar or close specifications to a 600$ GPU.

9 Conclusion

The `PALISADE` library offers a state-of-the-art implementation of RLWE-based Key Policy-Attribute-Based Encryption (KP-ABE) scheme that features computationally expensive homomorphic evaluation and encryption operations, which are amenable to hardware acceleration.

To this end, we presented a GPU implementation of the most expensive homomorphic operations in `PALISADE` that can support homomorphic evaluation and encryption of the KP-ABE scheme up to $l = 128$ attributes. Our GPU implementation is integrated within the main library and acts as a true accelerator, unburdening computationally demanding operations off the host CPU. We achieve significant acceleration figures for the main time-consuming operations of the KP-ABE scheme.

We, then, show that a KP-ABE-based publish/subscribe communication system, which supports end-to-end encryption and thus enhances the privacy of the communicating parties, is practicable. We envision and take advantage of a more realistic publish/subscribe scenario, whereby while published messages can assume many attributes, the access policies of the subscribers can be defined as logical functions over a fewer number of them.

One important factor in the acceleration is that our implementation does not use a special carrier prime, which would limit the sizes of the RNS primes that play an important factor in the performance of many homomorphic encryption schemes, including KP-ABE. Using RNS primes directly in the computation of number theoretic transform decreases the number of RNS primes and supports the continuation of arithmetic in the NTT domain. Our overall speedup figures confirm the advantage of our approach.

After the completion of our work, the developers of `PALISADE` released an extensible open-source post-quantum fully homomorphic encryption schemes library, which mainly contains the same API of `PALISADE` with additional modules and layers. More specifically, the Hardware Abstraction Layer (HAL) supports the integration of different hardware accelerators, such as FPGA and GPU. Therefore, our provided implementation in this would perform correctly with the newly introduced library with minor changes to the compiling scripts.

10 Future Work

Our work would urge other researchers to use it in further accelerations either of the *KP-ABE* or other homomorphic encryption schemes developed in `PALISADE`. On the other

hand, the GPU and its memory design are evolving in a way that we believe would significantly reduce the data communication overhead between the host and the device and increase the memory capacity, allowing larger policy circuits to execute faster on GPUs.

References

[1] Sahai, A., & Waters, B. (2005, May). Fuzzy identity-based encryption. In Annual international conference on the theory and applications of cryptographic techniques (pp. 457-473). Springer, Berlin, Heidelberg.

[2] Goyal, V., Pandey, O., Sahai, A., & Waters, B. (2006, October). Attribute-based encryption for fine-grained access control of encrypted data. In Proceedings of the 13th ACM conference on Computer and communications security (pp. 89-98).

[3] Attrapadung, N., Libert, B., & Panafieu, E. D. (2011, March). Expressive key-policy attribute-based encryption with constant-size ciphertexts. In International workshop on public key cryptography (pp. 90-108). Springer, Berlin, Heidelberg.

[4] Lai, J., Deng, R. H., Li, Y., & Weng, J. (2014, June). Fully secure key-policy attribute-based encryption with constant-size ciphertexts and fast decryption. In Proceedings of the 9th ACM symposium on Information, computer and communications security (pp. 239-248).

[5] R. Ostrovsky, A. Sahai, and B. Waters, "Attribute-based encryption with non-monotonic access structures," in Proc. CCS, Alexandria, VA, USA, Oct. 2007, pp. 195–203

[6] Han, J., Susilo, W., Mu, Y., & Yan, J. (2012). Privacy-preserving decentralized key-policy attribute-based encryption. IEEE transactions on parallel and distributed systems, 23(11), 2150-2162.

[7] Wang, C. J., & Luo, J. F. (2012, November). A key-policy attribute-based encryption scheme with constant size ciphertext. In 2012 Eighth International Conference on Computational Intelligence and Security (pp. 447-451). IEEE.

[8] Bethencourt, J., Sahai, A., & Waters, B. (2007, May). Ciphertext-policy attribute-based encryption. In 2007 IEEE symposium on security and privacy (SP'07) (pp. 321-334). IEEE.

[9] B. Waters, "Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization," in Proc. PKC, Taormina, Italy, Mar. 2011, pp. 53–70.

[10] J. Zhang, Z. Zhang, and A. Ge, "Ciphertext policy attribute-based encryption from lattices," in Proc. ASIACCS, Seoul, South Korea, May 2012, pp. 16–17.

[11] H. Deng et al., "Ciphertext-policy hierarchical attribute-based encryption with short ciphertexts," Inf. Sci., vol. 275, pp. 370–384, Aug. 2014.

[12] E. Zavattoni, L. J. D. Perez, S. Mitsunari, A. H. Sánchez-Ramírez, T. Teruya, and F. Rodríguez-Henríquez, "Software implementation of an attribute-based encryption scheme," IEEE Trans. Comput., vol. 64, no. 5, pp. 1429–1441, May 2015.

[13] Goyal, V., Jain, A., Pandey, O., & Sahai, A. (2008, July). Bounded ciphertext policy attribute based encryption. In International Colloquium on Automata, Languages, and Programming (pp. 579-591). Springer, Berlin, Heidelberg.

[14] Emura, K., Miyaji, A., Nomura, A., Omote, K., & Soshi, M. (2009, April). A ciphertext-policy attribute-based encryption scheme with constant ciphertext length. In International Conference on Information Security Practice and Experience (pp. 13-23). Springer, Berlin, Heidelberg.

[15] Cheung, L., & Newport, C. (2007, October). Provably secure ciphertext policy ABE. In Proceedings of the 14th ACM conference on Computer and communications security (pp. 456-465).

[16] D. Boneh et al., Fully key-homomorphic encryption, arithmetic circuit ABE and compact garbled circuits," in Proc. EUROCRYPT, Aarhus, Denmark, May 2014, pp. 533–556.

[17] Wei Dai, Berk Sunar, cuHE: A Homomorphic Encryption Accelerator Library. BalkanCryptSec 2015: 169-186

[18] Jean-Claude Bajard, Julien Eynard, Nabil Merkiche, Thomas Plantard, RNS Arithmetic Approach in Lattice-Based Cryptography: Accelerating the "Rounding-off" Core Procedure. ARITH 2015: 113-120

[19] Eugster, P. T., Felber, P. A., Guerraoui, R., & Kermarrec, A. M. (2003). The many faces of publish/subscribe. ACM computing surveys (CSUR), 35(2), 114-131.

[20] Dai, W., Doröz, Y., Polyakov, Y., Rohloff, K., Sajjadpour, H., Savaş, E., & Sunar, B. (2017). Implementation and evaluation of a lattice-based key-policy ABE scheme. IEEE Transactions on Information Forensics and Security, 13(5), 1169-1184.

[21] Polyakov, Y., Rohloff, K., & Ryan, G. W. (2017). Palisade lattice cryptography library user manual. Cybersecurity Research Center, New Jersey Institute ofTechnology (NJIT), Tech. Rep, 15.

[22] Borcea, C., Polyakov, Y., Rohloff, K., & Ryan, G. (2017). PICADOR: End-to-end encrypted Publish–Subscribe information distribution with proxy re-encryption. Future Generation Computer Systems, 71, 177-191.

[23] Polyakov, Y., Rohloff, K., Sahu, G., & Vaikuntanathan, V. (2017). Fast proxy re-encryption for publish/subscribe systems. ACM Transactions on Privacy and Security (TOPS), 20(4), 1-31.

[24] Bansarkhani, R. E., & Buchmann, J. (2013, August). Improvement and efficient implementation of a lattice-based signature scheme. In International Conference on Selected Areas in Cryptography (pp. 48-67). Springer, Berlin, Heidelberg.

[25] Genise, N., Micciancio, D., & Polyakov, Y. (2019, May). Building an efficient lattice gadget toolkit: Subgaussian sampling and more. In Annual International Conference on the Theory and Applications of Cryptographic Techniques (pp. 655-684). Springer, Cham.

[26] J. Fan and F. Vercauteren, Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch., vol. 2012, p. 144, 2012.

[27] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, (leveled) fully homomorphic encryption without bootstrapping, ACM Trans. Comput. Theory, vol. 6, no. 3, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2633600

[28] Jung Hee Cheon, Andrey Kim, Miran Kim, Yong Soo Song, Homomorphic Encryption for Arithmetic of Approximate Numbers. ASIACRYPT (1) 2017: 409-437

[29]

[30] C. Gentry, S. Halevi, and N. P. Smart, “Homomorphic evaluation of the AES circuit,” in Proc. CRYPTO, 2012, pp. 850–867.

[31] Ali Şah Özcan and Can Ayduman and Enes Recep Türkoğlu and Erkay Savaş, Homomorphic Encryption on GPU, Cryptology ePrint Archive, Paper 2022/1222, 2022.

[32] Ferraiolo, D.F. & Kuhn, D.R. (October 1992). "Role-Based Access Control". 15th National Computer Security Conference: 554–563.

[33] Sandhu, R., Coyne, E.J., Feinstein, H.L. and Youman, C.E. (August 1996). "Role-Based Access Control Models" (PDF). IEEE Computer. 29 (2): 38–47. CiteSeerX 10.1.1.50.7649. doi:10.1109/2.485845.

A Appendix: Algorithms

This section contains the algorithms mentioned throughout the paper in their order of mention.

Algorithm 1 `Encrypt`(M, $\mathcal{X}$, Master PKey)

Require: M: The message
Require: $\mathcal{X}$: The attributes set
Require: Master PKey, The master public key generated at the setup phase
Ensure: $\boldsymbol{C}_{in}, c_1$: The ciphertex vectors $\boldsymbol{C}_{in}$ can be partitioned into $\boldsymbol{C}_A$ and $\boldsymbol{C}_i$ for $i = 0, \ldots, \ell$

$s \leftarrow_U \mathcal{R}_q; e_1 \leftarrow \boldsymbol{D}_{\mathcal{R},\sigma}; \boldsymbol{e}_A \leftarrow \boldsymbol{D}_{\mathcal{R}^{1\times m},\sigma}$
$S_i \leftarrow_U \{\pm 1\}^{m\times m}$ where $i \in [0, l]$
$\boldsymbol{e}_0 \leftarrow (\boldsymbol{e}_A^T | \boldsymbol{e}_A^T S_0 | \boldsymbol{e}_A^T S_1 | \ldots | \boldsymbol{e}_A^T S_l)^T$
$\boldsymbol{C}_{in} \leftarrow (\boldsymbol{A} | (\boldsymbol{G} + \boldsymbol{B}_0) | (x_1 \boldsymbol{G} + \boldsymbol{B}_1) | (x_2 \boldsymbol{G} + \boldsymbol{B}_2) | \ldots | | (x_\ell \boldsymbol{G} + \boldsymbol{B}_\ell))^T \times s + \boldsymbol{e}_0$
$c_1 \leftarrow \beta s + e_1 + M \lceil \frac{q}{2} \rceil$
Return $(\boldsymbol{C}_{in}, c_1)$

Algorithm 2 `Evaluate`($\boldsymbol{C}_i$, $\boldsymbol{B}_i$, $\mathcal{X}$, ℓ')

Require: $\mathcal{C}$, $\mathcal{B}$: Ciphertext and public key vectors in the policy circuit
Require: x, ℓ': The attributes in the policy circuit and their cardinality
Ensure: $\boldsymbol{B}_{policy}, \boldsymbol{C}_{policy}$

for $(i = 1;\ i < \ell';\ i+ = 1)$ **do**
    $\bar{x}_{\ell'+i} \leftarrow (1 - \bar{x}_{2i-1}\bar{x}_{2i})$
    $\boldsymbol{\Psi}_i \leftarrow$ `NAFDECOMP`$(-\boldsymbol{B}_{2i-1})$
    $\boldsymbol{B}_{\ell'+i} \leftarrow \boldsymbol{B}_0 - \boldsymbol{B}_{2i}\boldsymbol{\Psi}_i$
    $\boldsymbol{C}_{\ell'+i} \leftarrow \boldsymbol{C}_0 - \bar{x}_{2i}\boldsymbol{C}_{2i-1} - \boldsymbol{\Psi}_i^T \boldsymbol{C}_{2i}$
end for
Return $\boldsymbol{B}_{2\ell'-1}, \boldsymbol{C}_{2\ell'-1}$

Algorithm 3 `EvalPK`($\mathcal{B}$, ℓ')

Require: $\mathcal{B}$: Public key vectors in the policy circuit
Require: x, ℓ': The attributes and their count
Ensure: $\boldsymbol{B}_{policy}$

for $(i = 1;\ i < \ell';\ i+ = 1)$ **do**
    $\boldsymbol{\Psi}_i \leftarrow$ `NAFDECOMP`$(-\boldsymbol{B}_{2i-1})$
    $\boldsymbol{B}_{\ell'+i} \leftarrow \boldsymbol{B}_0 - \boldsymbol{B}_{2i}\boldsymbol{\Psi}_i$
end for
Return $\boldsymbol{B}_{2\ell'-1}$

Algorithm 4 `EvalCT`($\mathcal{C}$, $\boldsymbol{B}_i$, $\mathcal{X}$, ℓ')

Require: $\mathcal{C}$: Ciphertext vectors in the policy circuit
Require: $\boldsymbol{B}_i$: Public key vectors, where $i \in [1, 2\ell' - 1]$
Require: x, ℓ': The attributes in the policy circuit and their cardinality
Ensure: $\boldsymbol{C}_{policy}$
1: **for** $(i = 1;\ i < \ell';\ i+ = 1)$ **do**
2: $\bar{x}_{\ell'+i} \leftarrow (1 - \bar{x}_{2i-1}\bar{x}_{2i})$
3: $\boldsymbol{\Psi}_i \leftarrow \texttt{NAFDECOMP}(-\boldsymbol{B}_{2i-1})$
4: $\boldsymbol{C}_{\ell'+i} \leftarrow \boldsymbol{C}_0 - \bar{x}_{2i}\boldsymbol{C}_{2i-1} - \boldsymbol{\Psi}_i^T \boldsymbol{C}_{2i}$
5: **end for**
6: **Return** $C_{2\ell'-1}$

Algorithm 5 Merge In-Place Forward NTT

Input: $a(x) \in \mathbb{Z}_q[x]/(x^n + 1)$ polynomial standard-order
Input: $\Omega_{br}[k] = \Omega^{br(k)}$ (Powers of the primitive root of unity ω sare tored in bit-reversed order)
Input: $n = 2^l$, q $(q \equiv 1 \mod 2n)$
Output: $\boldsymbol{a} \in \mathbb{Z}_q^n$ in bit-reversed order
1: $t = n;\quad m = 1$
2: **while** $m < n$ **do**
3: $t = t/2$
4: **for** i from 0 by 1 to m **do**
5: $j_1 = 2it$
6: $j_2 = j_1 + t - 1$
7: **for** j from j_1 by 1 to $j_2 + 1$ **do**
8: $U = a_j$
9: $V = a_{j+t} \cdot \Omega_{br}[m + i] \pmod q$
10: $a_j = U + V \pmod q$
11: $a_{j+t} = U - V \pmod q$
12: **end for**
13: **end for**
14: $m = 2m$
15: **end while**

Algorithm 6 (**NTT Kernel 2**) A GPU Algorithm for Merge in-Place Forward NTT when the number of threads in a block is greater than or equal to $n/2$

Input: $A[n], OmegaTable[n], n, q$
Output: $A[n]$
1: $idx = blockIdx.x \times blockDim.x + threadIdx.x$
2: **for** ℓ from 0 by 1 to $\log_2(n)$ **do**
3: $t = (n/2) \gg \ell$
4: $m = 2 \ll \ell$
5: $i = \lfloor \frac{idx}{t} \rfloor$
6: $address = i \cdot t + idx$
7: $U = A[address]$
8: $V = A[address + t]$
9: $\omega = OmegaTable[i + m]$
10: $V = (V \times \omega) \bmod q$
11: $A[address] = (U + V) \mod q$
12: $A[address + t] = (U - V) \bmod q$
13: **end for**

Algorithm 7 (**Kernel 1**)

Input: $A[n], OmegaTable[n], q, blockId, threadId$
Input: bc: no. of blocks ($bc = 2$) tc: no. of threads in a block
Output: $A[n]$

$idx = blockId \times tc + threadId$
$m = 1$; $t = n$
$k = n/(2 \times bc \times c)$
for i from 0 to $n/(2 \times tc\ )$ **do**
    $reg[i] = A[idx + (i \times (2 \times tc))]$
end for
for i from 0 to $\log_2 (n/(2 \times tc \times bc)) + 1$ **do**
    for j from 0 to $n/(tc \times 4)$ **do**
        $\ell = \lfloor \frac{j}{k} \rfloor \times k + j$
        $U = reg[\ell]$
        $V = reg[\ell + k]$
        $address = \lfloor \frac{idx}{t} \rfloor + m$
        $V = (V \times OmegaTable[address]) \mod q$
        $reg[\ell] = (U + V) \mod q$
        $reg[\ell + k] = (U - V) \mod q$
    end for
    $m = 2m$; $k = k/2$; $t = t/2$
end for
for i from 0 by 1 to $n/(tc \times 2)$ **do**
    $A[idx + (i \times (tc \times 2))] = reg[i]$
end for

B Appendix: Timing Results

Here, We provide the detailed results of our GPU implementation's run times on different hardware (GPU and CPU). In addition to the execution times of the publish/subscribe application, we integrated it into `PALISADE`.

The tables here include the detailed timings taken on three different GPUs of different architectures, compute capability, memory clock speeds, and bandwidths. The list of hardware specifications is provided in Table 2.

The Tables. 4–7 enumerate the running times of our GPU-based homomorphic encryption and evaluation operations in comparison with the running times on *Intel i7* and *Intel i9* processors.

Table 8 includes the timings of the homomorphic encryption and evaluation operations in the context of the proposed publish/subscribe application that allows encryption and evaluations with a different number of attributes. In other words, $l_{Enc} > l_{Eva}$ stands for the number of attributes/topics used in the encryption and evaluation, respectively.

The selected parameters in these trials indicate the minimal working set of parameters, including the number of attributes ℓ, the modulus q, the ring dimension n, and the base. Moreover, these parameters guarantee an acceptable level of security (at least 100-bit) and the implementation's correctness.

Table 4: Homomorphic evaluation operation (EvalCT & EvalPK) comparison with state-of-the-art CPU implementation in [25] (Timings in ms). The GPU timings include the kernels, CPU-GPU data copying, and the data copying from `PALISADE`'s data types into CUDA arrays.

Att. Count (ℓ)	$\log_2 q$	n	b	**GTX 1080**	**Quadro GV100**	**RTX 3070 Ti**	**CPU i7**[1]	**CPU i9**[2]
2	100 (50)	2^{12} (2^{11})	2^{20} (2^5)	14.06	19.34	6.31	23	23
4	100	2^{12}	2^{20}	36.13	51.78	18.15	72	72.6
8	150 (120)	2^{13}	2^{17} (2^{15})	360.6	518.3	141.03	590	525.2
16	200(180)	2^{13}	2^{20}	1548.27	1836.97	615.55	1680	1573.4
32	200(180)	2^{13}	2^{20} (2^{15})	3139.6	3768.32	1288.46	5670	5135.6
64	204	2^{13}	2^{15} (2^{17})	10249.88	12075.21	4141.05	13100	11954.8
128	300	2^{14}	2^{25}	11730.23	43121.81	4204.41	98300	55430.8

[1]: Intel Core i7-3770 CPU - 3.40GHz and 16GB of memory [25]

[2]: Intel Core i9-11900K CPU @ 3.50GHz and 32GB of memory.

Table 5: Homomorphic evaluation operation (EvalCT & EvalPK) timings in ms. The timings include the kernels operations and the NTT transformations.

Att. Count (ℓ)	$\log_2 q$	n	b	**GTX 1080**	**Quadro GV100**	**RTX 3070 Ti**
2	100 (50)	2^{12} (2^{11})	2^{20} (2^5)	8.78	12.42	6.03
4	100	2^{12}	2^{20}	21.49	29.54	17.36
8	150 (120)	2^{13}	2^{17} (2^{15})	172.72	188.01	77.47
16	200(180)	2^{13}	2^{20}	690.97	738.11	196
32	200 (180)	2^{13}	2^{20} (2^{15})	1416.71	1520.76	395.55
64	204	2^{13}	2^{15} (2^{17})	4267.88	4501.03	1309.68
128	300	2^{14}	2^{25}	7566.19	7717.74	2483.8

Table 6: Encrypt operation comparison with state-of-the-art CPU implementation in [25] (Timings in ms). The GPU timings include the kernels, CPU-GPU data copying, and the data copying from `PALISADE`'s data types into CUDA arrays.

Att. Count (ℓ)	$\log_2 q$	n	b	**GTX 1080**	**Quadro GV100**	**RTX 3070 Ti**	**CPU i7**[1]	**CPU i9**[2]
2	100 (50)	2^{12} (2^{11})	2^{20} (2^5)	17.73	38.10	11.46	7	4
4	100	2^{12}	2^{20}	19.42	37.77	11.71	15	5.2
8	150 (120)	2^{13}	2^{17} (2^{15})	87.64	268.77	41.8	56	18.8
16	200(180)	2^{13}	2^{20}	177.71	376.37	74.14	157	39.2
32	200(180)	2^{13}	2^{20} (2^{15})	248.78	445.71	93.83	414	78
64	204	2^{13}	2^{15} (2^{17})	536.54	816.24	176.11	1052	182.2
128	300	2^{14}	2^{25}	1561.5	2703.5	629.38	6454	1332

[1]: Intel Core i7-3770 CPU - 3.40GHz and 16GB of memory [25]

[2]: Intel Core i9-11900K CPU @ 3.50GHz and 32GB of memory.

Table 7: Encrypt operation timings in ms. It includes the kernels and the samplings from different distributions. The timings include the kernel executions and the sampling operations.

Att. Count (ℓ)	$\log_2 q$	n	$\boldsymbol{b}$	**GTX 1080**	**Quadro GV100**	**RTX 3070 Ti**
2	100 (50)	2^{12} (2^{11})	2^{20} (2^{5})	4.25	13.7	2.31
4	100	2^{12}	2^{20}	6.66	17.59	2.82
8	150 (120)	2^{13}	2^{17} (2^{15})	37.3	67.61	13.89
16	200(180)	2^{13}	2^{20}	105.89	151.21	32.92
32	200(180)	2^{13}	2^{20} (2^{15})	177.12	222.76	52.78
64	204	2^{13}	2^{15} (2^{17})	447.78	482.39	127.1
128	300	2^{14}	2^{25}	1283.41	2034.52	493.54

Table 8: Encrypt & Evaluation operations for the Pub/Sub application scenario (Timings in ms)

Att. Count (l_{Enc} / l_{Eva})	$\log_2 q$	n	$\boldsymbol{b}$	**Encrypt / Evaluation**		
				GTX1080	GV100	RTX3070 Ti
4/2	100	2^{12}	2^{20}	17.7 / 12.0	24.43 / 19.39	11.33 / 5.13
8/2	100	2^{12}	2^{20}	21.76 / 12.34	27.67 / 19.99	12.61 / 5.17
8/4	100	2^{12}	2^{20}	21.76 / 32.73	27.67 / 50.8	12.61 / 14.11
16/2	100	2^{12}	2^{20}	25.72 / 12.89	38.61 / 20.66	13.95 / 5.26
16/4	100	2^{12}	2^{20}	25.72 / 33.43	38.61 / 51.33	13.95 / 16.23
16/8	150	2^{13}	2^{25}	76.62 / 212.45	116.85 / 279.17	33.39 / 71.69
32/2	100	2^{13}	2^{20}	74.02 / 22.37	152.22 / 35.49	21.15 / 8.38
32/4	100	2^{13}	2^{20}	74.02 / 61.74	152.22 / 90.60	21.15 / 23.22
32/8	150	2^{13}	2^{25}	104.6 / 201.84	160.13 / 291.41	30.57 / 73.02
32/16	200	2^{13}	2^{25}	176.77 / 850.84	306.58 / 1042.93	54.43 / 336.52
64/2	100	2^{12}	2^{20}	65.66 / 13.05	114.8 / 18.53	21.85 / 9.3
64/4	100	2^{12}	2^{20}	65.66 / 36.11	114.8 / 50.07	21.85 / 22.68
64/8	150	2^{13}	2^{25}	177.66 / 205.47	291.31 / 271.76	31.27 / 72.56
64/16	200	2^{13}	2^{25}	275.95 / 845.77	426.12 / 1048.07	45.86 / 346.33
64/32	204	2^{13}	2^{15}	519.9 / 4925.22	778.03 / 6119.14	124.0 / 1961.02
128/2	100	2^{12}	2^{20}	124.38 / 13.57	167.34 / 19.32	21.41 / 8.46
128/4	100	2^{12}	2^{20}	124.38 / 34.08	167.34 / 49.74	21.41 / 23.02
128/8	150	2^{13}	2^{25}	301.1 / 200.23	408.9 / 288.69	30.99 / 78.08
128/16	204	2^{13}	2^{17}	697.03 / 1480.49	959.89 / 2086.63	224.0 / 650.67
128/32	204	2^{13}	2^{17}	697.03 / 3035.44	959.89 / 3907.83	224.0 / 1240.08
128/64	204	2^{13}	2^{15}	854.03 / 9674.46	1215.37 / 12542.97	280.82 / 4033.04