Partial and Fully Homomorphic Matching of IP Addresses Against Blacklists for Threat Analysis

William J Buchanan[1] and Hisham Ali[1]

Blockpass ID Lab, Edinburgh Napier University
b.buchanan@napier.ac.uk

Abstract. In many areas of cybersecurity, we require access to Personally Identifiable Information (PII), such as names, postal addresses and email addresses. Unfortunately, this can lead to data breaches, especially in relation to data compliance regulations such as GDPR. An IP address is a typical identifier which is used to map a network address to a person. Thus, in applications which are privacy-aware, we may aim to hide the IP address while aiming to determine if the address comes from a blacklist. One solution to this is to use homomorphic encryption to match an encrypted version of an IP address to a blacklisted network list. This matching allows us to encrypt the IP address and match it to an encrypted version of a blacklist. In this paper, we use the OpenFHE library [1] to convert network addresses into the BFV homomorphic encryption method. In order to assess the performance impact of BFV, it implements a matching method using the OpenFHE library and compares this against the partial homomorphic methods of Paillier, Damgard-Jurik, Okamoto-Uchiyama, Naccache-Stern and Benaloh. The main findings are that the BFV method compares favourably against the partial homomorphic methods in most cases.

1 Introduction

Data regulations such as GPDR [2] demand for greater control of Personally Identifiable Information (PII). In many areas of cybersecurity, we provide linkages between entities and their associated IP address and where revealing of an IP address can often identify a person or organisation that is involved in an investigation. With this, we could define a blacklist of networks that we need to identify if specific source IP address is included. One of the best ways to preserve privacy is with the use of homomorphic encryption, where we can encrypt both the target IP address and the blacklist and match them without revealing any additional information.

With homomorphic encryption, we can take plaintext of A and B and produce:

$$Enc_k(A \circ B) = Enc_k(A) \circ Enc_k(B) \tag{1}$$

and where $\circ$ can be any arithmetic operator such as add, subtract, multiply and divide. With symmetric key encryption, we use the same key to decrypt as we

do to encrypt. Overall, in analysing the IP matching problem, we need to either conduct bitwise homomorphic encryption or use a homomorphic subtraction method.

In the past, partial homomorphic methods (PHE) have been used within privacy-aware methods for network analysis. This includes Tusa *et al.*, who used the Paillier method to implement privacy-aware routing [3]. These methods often have fairly good performance levels, but they do not implement a full range of mathematical operations and thus often fall to scale on a large-scale basis, especially where the full range of operations is required.

This paper thus provides a new method for the usage of fully homomorphic encryption to match IP addresses to a blacklist of network addresses without revealing the IP address or the blacklist.

2 Fully Homomomorphic Encryption

Homomorphic encryption supports mathematical operations on encrypted data. In 1978, Rivest, Adleman, and Dertouzos [4] were the first to define the possibilities of implementing a homomorphic operation and used the RSA method. This supported multiply and divide operations [5], but does not support addition and subtraction. Overall, Partial Homomorphic Encryption (PHE) supports a few arithmetic operations, while Fully Homomorphic Encryption (FHE) supports add, subtract, multiply, and divide. Since Gentry defined the first FHE method [6] in 2009, there have been four main generations of homomorphic encryption:

- 1st generation: Gentry's method uses integers and lattices [7] including the DGHV method.
- 2nd generation. Brakerski, Gentry and Vaikuntanathan's (BGV) and Brakerski/ Fan-Vercauteren (BFV) use a Ring Learning With Errors approach [8]. The methods are similar to each other and with only a small difference between them.
- 3rd generation: These include DM (also known as FHEW) and CGGI (also known as TFHE) and support the integration of Boolean circuits for small integers.
- 4th generation: CKKS (Cheon, Kim, Kim, Song) and which uses floating-point numbers [9].

Generally, CKKS works best for real number computations and can be applied to machine learning applications as it can implement logistic regression methods and other statistical computations. DM (also known as FHEW) and CGGI (also known as TFHE) are useful in the application of Boolean circuits for small integers. BGV and BFV are generally used in applications with small integer values.

2.1 Public key or symmetric key

Homomorphic encryption can be implemented either with a symmetric key or an asymmetric (public) key. With symmetric key encryption, we use the same key

to encrypt as we do to decrypt, whereas, with an asymmetric method, we use a public key to encrypt and a private key to decrypt. In Figure 1 we use asymmetric encryption with a public key (pk) and a private key (sk). With this, Bob, Alice and Peggy will encrypt their data using the public key to produce ciphertext, and then we can operate on the ciphertext using arithmetic operations. The result can then be revealed by decrypting with the associated private key. We can also use symmetric key encryption (Figure 2), and where the data is encrypted with a secret key, and which is then used to decrypt the data. In this case, the data processor (Trent) should not have access to the secret key, as they could decrypt the data from the providers.

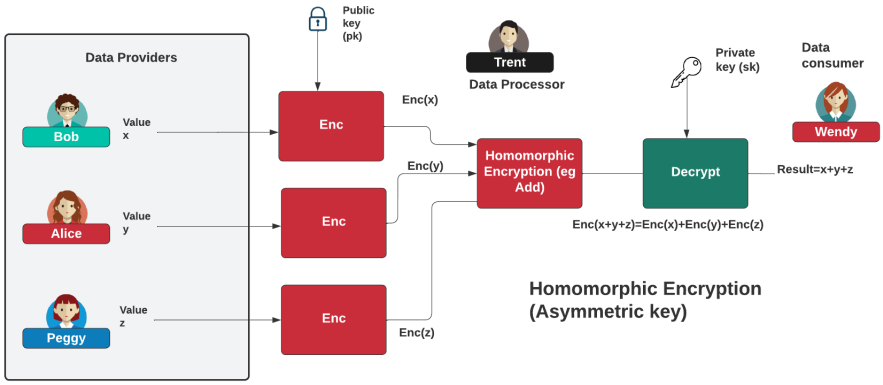


Fig. 1. Asymmetric encryption (public key)

2.2 Homomorphic libraries

There are several homomorphic encryption libraries that support FHe, including ones that support CUDA and GPU acceleration, but many have not been kept up-to-date with modern methods or have only integrated one method. Overall, the native language libraries tend to be the most useful, as they allow the compilation to machine code. The main languages for this are C++, Golang and Rust, although some Python libraries exist through wrappers to C++ code. This includes HEAAN-Python and its associated HEAAN library.

One of the first libraries which supported a range of methods are Microsoft SEAL [10], SEAL-C# and SEAL-Python. While it supports a wide range of methods, including BGV/BFV and CKKS, it has lacked any real serious development for the past few years. Wood et al. [11] define a full range of libraries. One of the most extensive libraries is PALISADE, and which has now developed into OpenFHE. Within OpenFHE, the main implementations are:

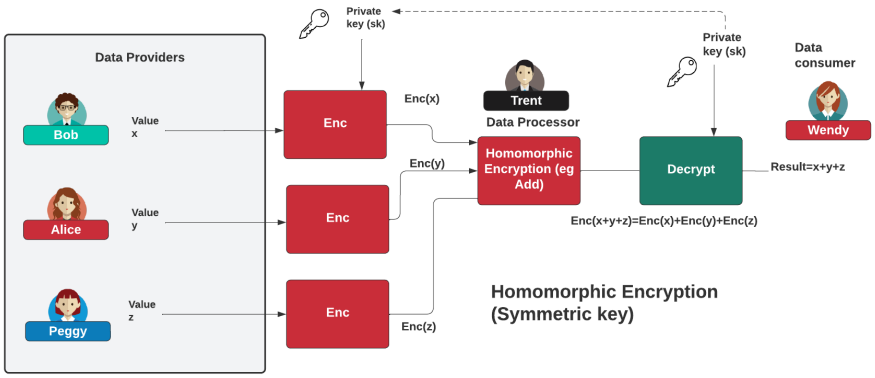


Fig. 2. Symmetric encryption

- Brakerski/Fan-Vercauteren (BFV) scheme for integer arithmetic.
- Brakerski-Gentry-Vaikuntanathan BGV) scheme for integer arithmetic
- Cheon-Kim-Kim-Song (CKKS) scheme for real-number arithmetic (includes approximate bootstrapping)
- Ducas-Micciancio (DM) and Chillotti-Gama-Georgieva-Izabachene (CGGI) schemes for Boolean circuit evaluation.

2.3 Bootstrapping

A key topic within fully homomorphic encryption is the usage of bootstrapping. Within a learning with-errors approach, we add noise to our computations. For a normal decryption process, we use the public key to encrypt data and then the associated private key to decrypt it. Within the bootstrap version of homomorphic encryption, we use an encrypted version of the private key that operates on the ciphertext. In this way, we remove the noise which can build up in the computation. Figure 3 outlines that we perform and evaluation on the decryption using an encrypted version of the private key. This will remove noise in the ciphertext, after which we can then use the actual private key to perform the decryption.

The main bootstrapping methods are CKKS [9], DM [12]/CGGI, and BGV/BFV. Overall, CKKS is generally the fastest bootstrapping method, while DM/CGGI is efficient with the evaluation of arbitrary functions. These functions approximate maths functions as polynomials (such as with Chebyshev approximation). BGV/BFV provides reasonable performance and is generally faster than DM/CGGI but slower than CKKS.

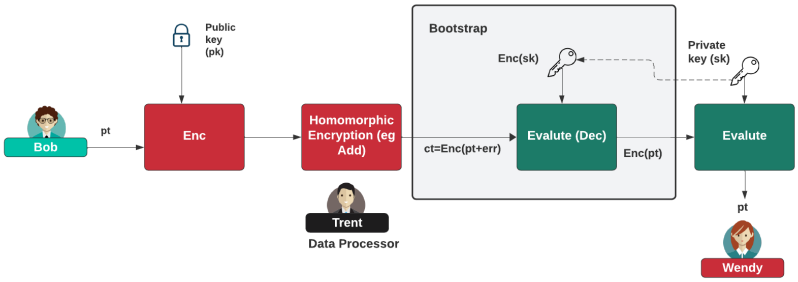


Fig. 3. Bootstrap

2.4 Plaintext slots

With many homomorphic methods, we can encrypt multiple plaintext values into ciphertext in a single operation. This is defined as the number of plaintext slots and is illustrated in Figure 4.

2.5 BGV and BFV

With BGV and BFV, we use a Ring Learning With Errors (LWE) method [8]. With BGV, we define a moduli (q), which constrains the range of the polynomial coefficients. Overall, the methods use a moduli, which can be defined within different levels. We then initially define a finite group of $\mathbb{Z}_q$ and then make this a ring by dividing our operations with $(x^n + 1)$ and where $n - 1$ is the largest power of the coefficients. The message can then be represented in binary as:

$$m = a_{n-1}a_{n-2}...a_0 \tag{2}$$

This can be converted into a polynomial with:

$$\mathbf{m} = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_0 \pmod{q} \tag{3}$$

The coefficients of this polynomial will then be a vector. Note that for efficiency, we can also encode the message with ternary (such as with -1, 0 and 1). We then define the plaintext modulus with:

$$t = p^r \tag{4}$$

and where p is a prime number and r is a positive number. We can then define a ciphertext modulus of q, and which should be much larger than t. To encrypt with the private key of $\mathbf{s}$, we implement:

$$(c_0, c_1) = \left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s} + e, -\mathbf{a}\right) \mod q \tag{5}$$

To decrypt:

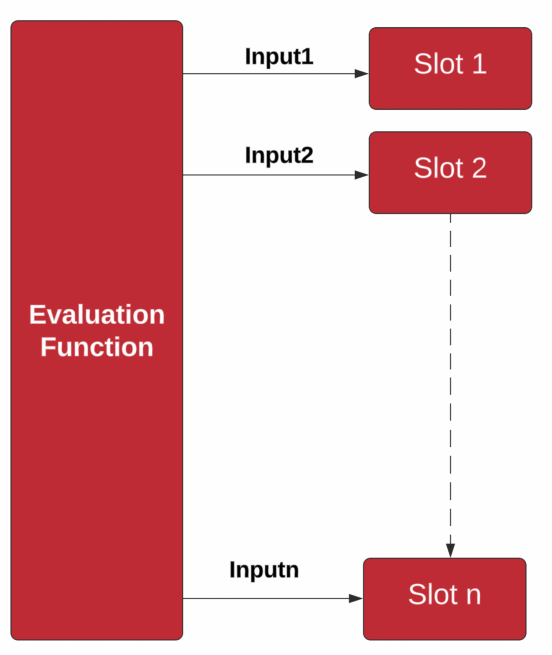


Fig. 4. Slots for plaintext

$$m = \lfloor \frac{t}{q}(c_0 + c_1).\mathbf{s} \rceil \tag{6}$$

This works because:

$$m_{recover} = \lfloor \frac{t}{q}\left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s} + e - \mathbf{a}.\mathbf{s}\right) \rceil \tag{7}$$

$$= \lfloor \left(\mathbf{m} + \frac{t}{q}.e\right) \rceil \tag{8}$$

$$\approx m \tag{9}$$

For two message of m_1 and m_2, we will get:

$$Enc(m_1 + m_2) = Enc(m_1) + Enc(m_2) \tag{10}$$

$$Enc(m_1.m_2) = Enc(m_1).Enc(m_2) \tag{11}$$

Noise and computation But each time we add or multiply, the error also increases. Thus, bootstrapping is required to reduce the noise. Overall, addition and plaintext/ciphertext multiplication is not a time-consuming task, but ciphertext/ciphertext multiplication is more computationally intensive. The most

computational task is typically the bootstrapping process, and the ciphertext/-ciphertext multiplication process adds the most noise to the process.

Parameters We thus have a parameter of the ciphertext modulus (q) and the plaintext modulus (t). Both of these are typically to the power of 2. An example of q is 2^{240} and for t is 65,537. As the value of 2^q is likely to be a large number, we typically define it as a log_q value. Thus, a ciphertext modulus of 2^{240} will be 240 as defined as a log_q value.

Public key generation We select the private (secret) key using a random ternary polynomial (-1, 0, and 0 coefficients) and which has the same degree as our ring. The public key is then a pair of polynomials as:

$$\mathbf{pk_1} = \mathbf{r}.\mathbf{sk} + e) \pmod{q} \tag{12}$$

$$\mathbf{pk_2} = \mathbf{r} \tag{13}$$

and where r is a random polynomial value. To encrypt with the public key (pk):

$$(c_0, c_1) = \left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s}.\mathbf{r} + e, -\mathbf{a}.\mathbf{r}\right) \mod q \tag{14}$$

We then decrypt with the private key (s);

$$m = \lfloor \frac{t}{q}(c_0 + c_1).\mathbf{s} \rceil \tag{15}$$

This works because:

$$m_{recovered} = \lfloor \frac{t}{q}\left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s}.\mathbf{r} + e - \mathbf{a}.\mathbf{r}.\mathbf{s}.\right) \rceil \tag{16}$$

$$= \lfloor \left(\mathbf{m} + \frac{t}{q}.e\right) \rceil \tag{17}$$

$$\approx m \tag{18}$$

3 Partial homomorphic encryption

With partial homomorphic encryption (PHE), we can implement some form of arithmetic operation in a homomorphic way. These methods include RSA, ElGamal, Paillier [13], Exponential ElGamal, Elliptic Curve ElGamal [14], Paillier [13], Damgard-Jurik [15], Okamoto–Uchiyama [16], Benaloh [17], Naccache–Stern [18], and Goldwasser–Micali [19]. Overall, we can use RSA and ElGamal for multiplicative homomorphic encryption; Paillier, Exponential ElGamal, Elliptic Curve ElGamal, Damgard-Jurik, Okamoto–Uchiyama, Benaloh and Naccache–Stern for additive homomorphic encryption; and Goldwasser–Micali for XOR homomorphic encryption.

3.1 Paillier

The Paillier cryptosystem [13] is a partial homomorphic encryption (PHE) method that can perform addition, subtraction, and scalar multiplying. Thus we get:

$$Enc_k(A+B) = Enc_k(A) + Enc_k(B) \tag{19}$$
$$Enc_k(A-B) = Enc_k(A) - Enc_k(B) \tag{20}$$
$$Enc_k(A.B) = A.Enc_k(B) \tag{21}$$

If we take two values: m_1 and m_2, we get two encrypted values of $Enc(m_1)$ and $Enc(m_2)$. We can then multiply the two cipher values to get $Enc(m_1+m_2)$. We can then decrypt to get $m_1 + m_2$. Along with this, we can also subtract to $Enc(m_1 - m_2)$. This is achieved by taking the inverse modulus of $Enc(m_2)$ and multiplying it with $Enc(m_1)$. Finally, we can perform a scalar multiply to get $Enc(m_1 \cdot m_2)$ and which is generated from $Enc(m1)^{m2}$.

First we select two large prime numbers (p and q) and compute:

$$N = pq \tag{22}$$
$$PHI = (p-1)(q-1) \tag{23}$$
$$\lambda = \text{lcm}\,(p-1, q-1) \tag{24}$$

and where lcm is the least common multiple. We then select a random integer g for:

$$g \in \mathbb{Z}^*_{nN^2} \tag{25}$$

We must make sure that n divides the order of g by checking the following:

$$\mu = (L(g^\lambda \pmod n)^2))^{-1} \pmod N \tag{26}$$

and where L is defined as $L(x) = \frac{x-1}{N}$. The public key is (N, g) and the private key is $\lambda, \mu)$.

To encrypt a message (M), we select a random r value and compute the ciphertext of:

$$c = g^m \cdot r^N \pmod{N^2} \tag{27}$$

and then to decrypt:

$$m = L(c^\lambda \pmod N)^2) \cdot \mu \pmod N \tag{28}$$

For adding and scalar multiplying, we can take two ciphers (C_1 and C_2), and get:

$$C_1 = g^{m_1} \cdot r_1^N \pmod{N^2} \tag{29}$$
$$C_2 = g^{m_2} \cdot r_2^N \pmod{N^2} \tag{30}$$

If we now multiply them, we get:

$$C_1 \cdot C_2 = g^{m_1} \cdot r_1^N \cdot g^{m_2} \cdot r_2^N \pmod{N^2} \tag{31}$$

$$C_1 \cdot C_2 = g^{m_1+m_2} \cdot r_1^N \cdot r_2^N \pmod{N^2} \tag{32}$$

Adding two values requires the multiplication of the ciphers. Examples of using Paillier are defined at [20].

3.2 Benaloh

In 1986, Josh (Cohen) Benaloh published on *A Robust and Verifiable Cryptographically Secure Election Scheme* [17,21]. Within it, Josh outlined a public key encryption method and where Bob could generate a public key and a private key. The public key could then be used by Alice to encrypt data for Bob, and then Bob can use the associated private key to decrypt it. It has the advantage of supporting additive homomorphic encryption. For the Benaloh method, to generate a key pair, Bob generates p and q, and which are two large prime numbers, which are two large distinct prime numbers. Next, he computes:

$$n = pq \tag{33}$$

$$\phi(n) = (p–1)(q–1) \tag{34}$$

Bob then selects a block size (r) so that:

- r divides $p-1$
- $textrmgcd(r,(p-1)/r) = 1$
- $textrmgcd(r,q-1) = 1$

Next Bob selects y so that:

$$x = y^{\phi(n)/r} \pmod{n} \neq 1 \tag{35}$$

Bob's private key is (p,q) and his public key is (y,r,n). To encrypt data for Bob, Alice selects a message (m) and uses Bob's public key of (y,r,n). First, she selects a random value of u and which is between 0 and n. Alice then encrypts with:

$$c = y^m u^r \pmod{n} \tag{36}$$

She sends this ciphertext to Bob. He will then decrypt with:

$$a = c^{\phi(n)/r} \pmod{n} \tag{37}$$

Bob lets $md = 0$. If $x^{md} \pmod{n} \neq a$ the Bob increments md by 1. He keeps doing this until:

$$x^{md} \pmod{n} = a \tag{38}$$

The value of md is then the original plaintext. One of the advantages of the Benaloh method is that we can perform additive homomorphic encryption. If we have two messages, we multiply the ciphers together for each message:

$$c_1 = y^{m_1} u^r \pmod{n} \tag{39}$$
$$c_2 = y^{m_2} u^r \pmod{n} \tag{40}$$
$$c = c_1.c_2 \tag{41}$$

This will give:

$$c = y^{m_1+m_2} u^r \pmod{n} \tag{42}$$

3.3 Okamoto-Uchiyama

With the Okamoto-Uchiyama method [16,22], we can perform additive and scalar multiply homomorphic encryption. A public/private key pair is generated as follows:

- Generate large primes p and q and set $n = p^2 q$.
- $g \in (\mathbb{Z}/n\mathbb{Z})^*$ such that $g^{p-1} \neq 1 \mod p^2$.
- Let $h = g^n \pmod{n}$.

The public key is (n, g, h) and then the private key is (p, q). To encrypt a message m, where m is taken to be an element in 2^{k-1}. We then select $r \in \mathbb{Z}/n\mathbb{Z}$ at random. The cipher is then:

$$C = g^m h^r \pmod{n} \tag{43}$$

Next, we define the function of: $L(x) = \frac{x-1}{p}$.
We then decrypt with:

$$m = \frac{L\left(C^{p-1} \mod p^2\right)}{L\left(g^{p-1} \mod p^2\right)} \pmod{p} \tag{44}$$

3.4 Naccache–Stern

With the Naccache–Stern method [18,23], we select a large prime number (p). We then select a value (n) and for i from 0 to n, we select the the first n prime numbers $(p_0 ... p_{n-1}$ of which p_0 is 2. We must make sure that:

$$\prod_{i=0}^{n} p_i lt; p \tag{45}$$

For our secret key (s) we make sure that:

$$gcd(s, p-1) = 1 \tag{46}$$

To compute the public key (v_i), we calculate: $v_i = \sqrt[s]{p_i} \pmod p$ To encrypt, we take a message of$m$ and then determine the message bits ofm_i. We can then cipher with the public key:

$$c = \prod_{i=0}^{n} v_i^{m_i} \pmod p \tag{47}$$

and then to decrypt:

$$m = \sum_{i=0}^{n} \frac{2^i}{p_i - 1} \times (\gcd(p_i, c^s \pmod p) - 1) \tag{48}$$

3.5 Goldwasser–Micali

With public key encryption, Alice could have two possible messages (a '0' or a '1') that she sends to Bob. If Eve knows the possible messages (a '0' or a '1'), she will then cipher each one with Bob's public key and then matches the results against the cipher message that Alice sends. Eve can thus determine what Alice has sent to Bob. In order to overcome this, the Goldwasser–Micali (GM) method is used as a public key algorithm that uses a probabilistic public-key encryption scheme. In this case, we will implement an XOR homomorphic encryption operation. For an input of 17 (10001) and 16 (10000), we will get a result of 00001 (1).

In a probabilist encryption method, Alice selects the plaintext (m) and a random string (r). Next, she uses Bob's public key to encrypt the message pair of (m, r). If the value is random, then Eve will not be able to use the range of messages and random values used. If Bob wants to create his public and private keys. He first selects two random prime numbers for his private key and then calculates N:

$$N = pq \tag{49}$$

The values of p and q will be his private key, and N will form part of his public key. For the second part of his public key, he determines:

$$a = \text{pseudosquare}(p, q) \tag{50}$$

For this, we determine if we can find, for a given value of a, which has no solutions:

$$u^2 \equiv a \pmod{p} \tag{51}$$

$$u^2 \equiv a \pmod{q} \tag{52}$$

This means that there are no quadratic residues. Bob's public key is (N, a) and the private key is (p, q). The key encryption method becomes:

- Bob selects p and q.
- Bob selects a with $\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = -1$. This is a Jacobi symbol calculation.
- Bob publishes N and a.

To encrypt for Bob:

- Select a bit to encrypt $m \in 0, 1$.
- Alice uses Bob's values of $N, a)$ to compute:
- $c = r^2 \pmod{N}$ if $m = 0$
- $c = ar^2 \pmod{N}$ if $m = 1$

Alice chooses r at random, and thus, Eve will not be able to spot the message, as the random values will consist of all possible squares modulo N when $m = 0$. Alice sends c to Bob. To decrypt, Bob then computes $\left(\frac{c}{p}\right)$ and gets:

$$m = 0 : if \left(\frac{c}{p}\right) = 1 \tag{53}$$

$$m = 1 : if \left(\frac{c}{p}\right) = -1 \tag{54}$$

3.6 Damgard-Jurik

With the Damgard-Jurik method [15] we select two large prime numbers (p and q) and compute:

$$n = pq \tag{55}$$

$$\phi = (p-1)(q-1) \tag{56}$$

$$\lambda = \operatorname{lcm} \phi \tag{57}$$

and where lcm is the least common multiple. We then select a random integer g for:

$$g \in \mathbb{Z}^*_{n^2} \tag{58}$$

We must make sure that n divides the order of g by checking the following:

$$\mu = (L(g^\lambda \bmod n^2))^{-1} \bmod n \tag{59}$$

and where L is defined as $L(x) = \frac{x-1}{n}$. The public key is (n, g) and the private key is $\lambda, \mu)$. To encrypt a message (M), we select a random r value and compute the ciphertext of:

$$c = g^m \cdot r^n \pmod{n^{s+1}} \tag{60}$$

and then to decrypt:

$$m = L(c^\lambda \mathrm{mod} n^2) \cdot \mu \pmod{n}^s \tag{61}$$

If we take two ciphers (C_1 and C_2), we get:

$$C_1 = g^{m_1} \cdot r_1^n \pmod{n^2} \tag{62}$$
$$C_2 = g^{m_2} \cdot r_2^n \pmod{n^2} \tag{63}$$

If we now multiply them, we get:

$$C_1 \cdot C_2 = g^{m_1} \cdot r_1^n \cdot g^{m_2} \cdot r_2^n \pmod{n^2} \tag{64}$$
$$C_1 \cdot C_2 = g^{m_1+m_2} \cdot r_1^n \cdot r_2^n \pmod{n^2} \tag{65}$$

Adding two values requires multiplication of the ciphers. If we now divide them, we get:
$\frac{C_1}{C_2} = \frac{g^{m_1} \cdot r_1^n}{g^{m_2} \cdot r_2^n} \pmod{n^2}$ $\frac{C_1}{C_2} = g^{m_1 - m_2} \frac{r_1^n}{r_2^n} \pmod{n^2}$
Thus, subtraction is equivalent to a divide operation. For this, we perform a modular divide operation.

4 Methodology

An IPv4 address has four main fields that are defined with integer values in the range 0-255. An example is 12.23.45.67, and which is a 32-bit address value, and where each of the fields is identified with an 8-bit value. The address then splits into a network part and a host part, such as where 12.23.46.0 might identify a network address, and 0.0.0.67 will identify the host part. Overall, we define the network part with a subnet mask, and where bits that are set to a 1 identifies the network part, and where we have a 0, we define the host part. We can then vary the number of 1's from 0 to 32. An example subnet mask where the network part is 24 bits long is 0xffffff00, and which can be represented by 255.255.255.0. This is often identified by the number of bits, such as 1's in the subnet mask, such as '/24'. An example IP address could thus be '192.168.0.10/24', and where the network part is '192.168.0.0" and the host part of '0.0.0.24'.

4.1 PHE Subtraction method

Algorithm 4.1 defines the method used for subtractive homomorphic encryption. In this, we convert the target IP address and the blacklisted network address to 32-bit integer values. The subnet mask then defines the network part to match. Again, this will be an integer value, and where the 1's identify the network part and the 0's will identify the host part. Overall, we are only interested in matching the network part of the target address to the blacklisted network address. We can then create a BFV keypair with a public key (pk) and a private key (sk). The homomorphic public key is then used to encrypt the target IP address and also the blacklisted address. Once encrypted, we can then perform a homomorphic subtraction. If the network part of the target IP address and the network address match, the result will be an encrypted value of zero. We can then decrypt the result of the homomorphic subtraction and if we get a zero, we know that the IP address is contained in the blacklist.

Algorithm 1 FHE for IP detection

Set IP with the address to find for an integer
Set $Network$ for the blacklisted addresses as an integer
Set $Subnet_{mask}$ as the subnet of blacklist
Generate pk, sk for homomorphic key pair
$IP_e = Enc(IP, pk)$
$Blacklist = Network \wedge Subnet_{mask}$
$Blacklist_e = Enc(Blacklist, pk)$
$Enc_{diff} = IP_e - Blacklist_e$
if $Enc_{diff} = 0$ **then**
    Address is in the blacklist
else
    Address is not in the blacklist
end if

4.2 Goldwasser–Micali XOR method

For the Goldwasser–Micali partial homomorphic encryption method, we can use the XOR operation, and where we can simply XOR the blacklist network address with the network address of the target IP address. This method is defined in Equation 4.2.

4.3 OpenFHE parameters

The parameters that need to be set within OpenFHE for BFV are:

- *Scheme*. This defines the scheme to be used. In the case of BFV, this is set to BFVRNS_SCHEME.

Algorithm 2 FHE for IP detection

Set IP with the address to find for an integer
Set $Network$ for the blacklisted addresses as an integer
Set $Subnet_{mask}$ as the subnet of blacklist
Generate pk, sk for homomorphic key pair
$IP_e = Enc(IP, pk)$
$Blacklist = Network \wedge Subnet_{mask}$
$Blacklist_e = Enc(Blacklist, pk)$
$Enc_{diff} = IP_e \oplus Blacklist_e$
if $Enc_{diff} = 0$ **then**
    Address is in the blacklist
else
    Address is not in the blacklist
end if

- $RDim$. This defines the size of the lattice ring dimension. A typical value for this is 16,384.
- $MultDepth$. This is the multiplication depth and is the maximum number of sequential (cascaded) multiplications that are performed on encrypted data before decryption fails due to excessive noise accumulation.
- $PtMod$. This defines the plaintext modulus, and needs to be a prime number which is larger than the number of bits in the plaintext.

The parameter set needs to support integer values up to 32 bits. A recommended setup for 41-bit resolution for the plaintext ($PtMod$) is set at 35,184,372,744,193, along with the other parameters defined at [24]. The following section defines the implementation of the method.

5 Results

The coding for fully homomorphic encryption using OpenFHE is defined in the Coding section. The results for a t3.medium instance on AWS is given in Table 5, and which include a comparison with partial homomorphic methods using the PHE Library [25]. The time to set up the key pair and the context for the encryption is measured at an average of 93 ms. The greatest overhead is then the time it takes to encrypt the values, which has an average time of around 270 ms. The subtraction and decryption timing then comes in around 16 ms. It can be seen that the encryption process provides the largest overhead in IP address matching. We can see that the Benaloh and Goldwasser-Micali methods are by far the fastest. The BFV method has a comparable performance as compared with the PHE methods and is actually faster in the homomorphic encryption operation than Paillier, Damgard-Jurik and Okamoto-Uchiyama. The Naccache-Stern method also performs well, especially in the encryption and decryption process. Overall, it is the encryption process that tends to have the greatest overhead in processing.

Table 1: Results for homomorphic operations for IP matching

Method	Key pair (ms)	Encrypt (ms)	Operation and decrypt (ms)
BFV (OpenFHE) [1]	93.1	270.2	16.4
Paillier [13]	55.7	168.6	66.3
Damgard-Jurik [15]	83.6	443.5	155.2
Okamoto-Uchiyama [16]	128.9	206.9	18.0
Naccache-Stern [18]	48.2	0.2	0.6
Benaloh	6.0	0.3	0.2
Goldwasser-Micali [19]	0.3	0.5	1.6

6 Conclusion

The increasing requirements for privacy-aware cybersecurity provide opportunities to encrypt data using homomorphic encryption. This paper outlines a method that requires plaintext to support 32 bits and requires a larger plaintext modulus than is used by default in applications. The paper has thus outlined a method that uses the popular OpenFHE library and has fairly reasonable overheads in latency in creating the homomorphic encryption keys and in encrypting, processing, and decrypting data.

7 Coding

The coding required for IP matching using the OpenFHE library is [26]:

```
#include <openfhe.h>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
using namespace lbcrypto;
#include <sstream>

unsigned long hex2dec(string hex)
{
    unsigned long result = 0;
    for (int i=0; i<hex.length(); i++) {
        if (hex[i]>=48 && hex[i]<=57)
        {
            result += (hex[i]-48)*pow(16,hex.length()-i-1);
        } else if (hex[i]>=65 && hex[i]<=70) {
            result += (hex[i]-55)*pow(16,hex.length( )-i-1);
        } else if (hex[i]>=97 && hex[i]<=102) {
            result += (hex[i]-87)*pow(16,hex.length()-i-1);
        }
    }
    return result;
}
```

```
// https://stackoverflow.com/questions/5328070/how-to-convert-string-to-
    ip-address-and-vice-versa
uint32_t convert( const std::string& ipv4Str )
{
    std::istringstream iss( ipv4Str );
    uint32_t ipv4 = 0;
    for( uint32_t i = 0; i < 4; ++i ) {
        uint32_t part;
        iss >> part;
        if ( iss.fail() || part > 255 ) {
            throw std::runtime_error( "Invalid␣IP␣address␣-␣Expected␣[0,␣
                255]" );
        }
        // LSHIFT and OR all parts together with the first part as the MSB
        ipv4 |= part << ( 8 * ( 3 - i ) );

        // Check for delimiter except on last iteration
        if ( i != 3 ) {
            char delimiter;
            iss >> delimiter;
            if ( iss.fail() || delimiter != '.' ) {
                throw std::runtime_error( "Invalid␣IP␣address␣-␣Expected␣
                    '.'␣delimiter" );
            }
        }
    }
    return ipv4;
}

int main(int argc, char* argv[]) {
uint64_t mod=35184372744193;

    string ip1="2.3.4.5";
    string network_address="2.3.4.7";
    uint32_t subnet_mask=0xffffff00;

    uint32_t ipval = convert(ip1) & subnet_mask;
    uint32_t network = (convert(network_address)) & subnet_mask;

    CCParams<CryptoContextBFVRNS> parameters;
    parameters.SetPlaintextModulus(mod);
    parameters.SetMultiplicativeDepth(0);

    CryptoContext<DCRTPoly> cryptoContext = GenCryptoContext(parameters);
    cryptoContext->Enable(PKE);
    cryptoContext->Enable(KEYSWITCH);
    cryptoContext->Enable(LEVELEDSHE);

    KeyPair<DCRTPoly> keyPair;
```

```
    // Generate key pair
    keyPair = cryptoContext->KeyGen();

    std::vector<int64_t>xval = {1};
        xval[0]=ipval;
    Plaintext xplaintext = cryptoContext->MakePackedPlaintext(xval);

    std::vector<int64_t> yval = {1};
        yval[0]=network;
    Plaintext yplaintext = cryptoContext->MakePackedPlaintext(yval);

    // Encrypt values
    auto ciphertext1 = cryptoContext->Encrypt(keyPair.publicKey,
        xplaintext);
    auto ciphertext2 = cryptoContext->Encrypt(keyPair.publicKey,
        yplaintext);

    // Subtract ciphertext
    auto ciphertextMult = cryptoContext->EvalSub(ciphertext1, ciphertext2
        );

    // Decrypt result
    Plaintext plaintextAddRes;
    cryptoContext->Decrypt(keyPair.secretKey, ciphertextMult, &
        plaintextAddRes);

    std::cout << "Modulus:␣:␣" << mod<< std::endl;

    std::cout << "\nIP1:␣" << xplaintext << std::endl;
    std::cout << "IP2:␣" << yplaintext << std::endl;

    // Output results
    std::cout << "\nDifference" << std::endl;

    plaintextAddRes->SetLength(1);
    auto res = plaintextAddRes->GetPackedValue();
    std::cout << "Subnet␣test=␣" << res[0] << std::endl;

    if (res[0]==0) std::cout << "IP␣address␣is␣in␣subnet" << std::endl;
    else std::cout << "IP␣address␣is␣not␣in␣the␣subnet" << std::endl;

    return 0;
}
```

References

1. W. J. Buchanan, “Homomorphic encryption (openfhe),” https://asecuritysite.com/openfhe, Asecuritysite.com, 2025, accessed: February 20, 2025. [Online]. Available: https://asecuritysite.com/openfhe

2. G. GDPR, "General data protection regulation," *Regulation (EU)*, vol. 679, 2016.
3. F. Tusa, D. Griffin, and M. Rio, "Homomorphic routing: Private data forwarding in the internet," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing*, 2023, pp. 1–7.
4. R. L. Rivest, L. Adleman, M. L. Dertouzos *et al.*, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.
5. W. J. Buchanan, "Openfhe," https://github.com/openfheorg/openfhe-development, OpenFHE, 2024, accessed: Feb 20, 2025. [Online]. Available: https://github.com/openfheorg/openfhe-development
6. C. Gentry, "A fully homomorphic encryption scheme," 2009, crypto.stanford.edu/craig.
7. M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*. Springer, 2010, pp. 24–43.
8. Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," *SIAM Journal on computing*, vol. 43, no. 2, pp. 831–871, 2014.
9. J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 2017, pp. 409–437.
10. W. J. Buchanan, "Homomorphic encryption (seal)," https://asecuritysite.com/seal, Asecuritysite.com, 2024, accessed: September 04, 2024. [Online]. Available: https://asecuritysite.com/seal
11. A. Wood, K. Najarian, and D. Kahrobaei, "Homomorphic encryption for machine learning in medicine and bioinformatics," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–35, 2020.
12. L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 617–640.
13. P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1999, pp. 223–238.
14. T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.
15. I. Damgård, M. Jurik, and J. B. Nielsen, "A generalization of paillier's public-key system with applications to electronic voting," *International Journal of Information Security*, vol. 9, pp. 371–385, 2010.
16. T. Okamoto and S. Uchiyama, "A new public-key cryptosystem as secure as factoring," in *Advances in Cryptology—EUROCRYPT'98: International Conference on the Theory and Application of Cryptographic Techniques Espoo, Finland, May 31–June 4, 1998 Proceedings 17*. Springer, 1998, pp. 308–318.
17. J. D. Cohen and M. J. Fischer, *A robust and verifiable cryptographically secure election scheme*. Yale University. Department of Computer Science, 1985.
18. D. Naccache and J. Stern, "A new public-key cryptosystem," in *Advances in Cryptology—EUROCRYPT'97: International Conference on the Theory and Applica-*

tion of Cryptographic Techniques Konstanz, Germany, May 11–15, 1997 Proceedings 16. Springer, 1997, pp. 27–36.
19. S. Goldwasser and S. Micali, "Probabilistic encryption & how to play mental poker keeping secret all partial information," in *Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 173–201.
20. W. J. Buchanan, "Paillier - partial homomorphic encryption," https://asecuritysite.com/paillier/, Asecuritysite.com, 2024, accessed: September 03, 2024. [Online]. Available: https://asecuritysite.com/paillier/
21. ——, "Benaloh public key encryption," https://asecuritysite.com/homomorphic_partial/benaloh2, Asecuritysite.com, 2025, accessed: February 21, 2025. [Online]. Available: https://asecuritysite.com/homomorphic_partial/benaloh2
22. ——, "Okamoto uchiyama public key encryption," https://asecuritysite.com/homomorphic_partial/ou_homomorphic, Asecuritysite.com, 2025, accessed: February 21, 2025. [Online]. Available: https://asecuritysite.com/homomorphic_partial/ou_homomorphic
23. ——, "Naccache stern public key encryption," https://asecuritysite.com/homomorphic_partial/nacc_enc, Asecuritysite.com, 2025, accessed: February 21, 2025. [Online]. Available: https://asecuritysite.com/homomorphic_partial/nacc_enc
24. OpenFHE, "Openfhe unit tests," https://github.com/openfheorg/openfhe-development/blob/main/src/pke/unittest/UnitTestSHE.cpp, OpenFHE.com, 2025, accessed: September 06, 2024. [Online]. Available: https://github.com/openfheorg/openfhe-development/blob/main/src/pke/unittest/UnitTestSHE.cpp
25. S. I. Serengil, "Phe library," https://github.com/serengil/LightPHE, PHE, 2025, accessed: February 21, 2025. [Online]. Available: https://github.com/serengil/LightPHE
26. OpenFHE, "Ip marking," https://shorturl.at/HAwt4, OpenFHE.com, 2025, accessed: Feb 20, 2025. [Online]. Available: https://shorturl.at/HAwt4