Hardware Assisted Fully Homomorphic Function Evaluation and Encrypted Search

Project Report

Sujoy Sinha Roy[1], Frederik Vercauteren[1,2], Jo Vliegen[1], and Ingrid Verbauwhede[1]
[1] ESAT/COSIC and iMinds, KU Leuven,
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
[2] Open Security Research,
FangDa Building 704, Kejinan-12th, Nanshan, Shenzhen, 518000, China
email: {firstname.lastname}@esat.kuleuven.be

Abstract

In this report we propose a scheme to perform homomorphic evaluations of arbitrary depth with the assistance of a special module *recryption box*. Existing somewhat homomorphic encryption schemes can only perform homomorphic operations until the noise in the ciphertexts reaches a critical bound depending on the parameters of the homomorphic encryption scheme. The classical approach of bootstrapping also allows for arbitrary depth evaluations, but has a detrimental impact on the size of the parameters, making the whole setup inefficient. We describe two different instantiations of our recryption box for assisting homomorphic evaluations of arbitrary depth. The recryption box refreshes the ciphertexts by lowering the inherent noise and can be used with any instantiation of the parameters, i.e. there is no minimum size unlike bootstrapping.

To demonstrate the practicality of the proposal, we design the recryption box on a Xilinx Virtex 6 FPGA board ML605 to support the FV somewhat homomorphic encryption scheme. The recryption box requires 0.43 ms to refresh one ciphertext. Further, we use this recryption box to boost the performance of encrypted search operation. On a 40 core Intel server, we can perform encrypted search in a table of 2^{16} entries in around 20 seconds. This is roughly 20 times faster than the implementation without recryption box.

Keywords. Homomorphic encryption, FV, lattice-based cryptography, ring-LWE, polynomial multiplication, number theoretic transform, hardware implementation

I. Introduction

For many years the construction of a fully homomorphic encryption (FHE) scheme was an open problem in cryptography. FHE enables computations on encrypted data without the need for decryption and a practical realization of FHE would allow users to outsource computations to an untrusted cloud server. In 2009 Gentry [12] constructed the first fully homomorphic encryption (FHE) scheme by using ideal lattices. Gentry's FHE scheme uses a somewhat homomorphic encryption scheme (SHE) combined with a mechanism known as *bootstrapping*. An SHE scheme can be used to compute on the encrypted data, but each operation increases the noise inherent in the ciphertexts. Once the noise reaches a certain threshold that depends on the parameters of the scheme, decryption will fail. The bootstrapping operation is used to publicly "refresh" a noisy ciphertext and repeated application enables evaluations of arbitrary depth. However, bootstrapping is only possible if the parameters of the SHE are chosen large enough to accommodate for the bootstrapping operation, thereby also slowing down the actual function evaluation.

Though Gentry's scheme offered homomorphic function evaluation of any depth, the performance of the scheme is really impractical. Since 2010 many researchers have improved the performance of FHE [3], [4], [7], [26], [11], [13], [14], [18] by using the (ring) learning with errors (ring-LWE) problem or the NTRU problem. Though the performance of FHE schemes has improved orders of magnitude compared to Gentry's first FHE scheme, their practicality still remains rather low. The main problem is that the bootstrapping operation is tremendously slow, e.g. for [6] it takes 172 seconds on an Intel Core-i7 processor. Hence it is not yet possible to deploy FHE in cloud computations. Even somewhat homomorphic encryption schemes that can evaluate functions of small complexity take a large amount time. For e.g., evaluation of one SIMON-64/128 decryption on encrypted data takes more than an hour on a 4-core Intel Core-i7 processor [16].

Our solution is to bypass this costly bootstrapping operation using a third party *recryption box* that is instantiated in two different setups. In the first setup the recryption box uses a key switching technique that allows the cloud server to convert a ciphertext encrypted under user's public key into a ciphertext encrypted under box's public key. With this the box performs a decryption using its own private key and then a re-encryption using user's public key. Naturally the large noise in the encrypted data is eliminated. In the second setup a multiparty computation scheme is used: noisy ciphertexts are decrypted among multiple parties, and then reencrypted again. This re-encryption operation gives a freshly encrypted data with limited noise as the shared multiparty decryption operation removes the large noise inherent in the ciphertexts. During the execution

of an application on encrypted data, the cloud performs the homomorphic operations, and then sends the dirty ciphertexts to the third party recryption box (or boxes). Although in this report we instantiate the recryption box on an FPGA, we note it is possible to implement it in a trusted execution environment or using specialized instructions such as SGX, assuming sufficient countermeasures are taken against physical attacks.

In the field of cryptographic implementations, hardware accelerators have been used to speedup cryptographic computations. Since the existing homomorphic schemes are very costly, several hardware architectures to speed up homomorphic evaluations have been proposed. In [10] an ASIC implementation of the Gentry-Halevi FHE scheme was presented. In a 90 nm CMOS technology, the FHE architecture computes encryption, decryption, and bootstrapping operations in 18.1ms, 16.1ms, and 3.1s respectively and consumes less than 30 million gates. The architecture uses a number theoretic transform (NTT) based million bit multiplier to speedup computation. We see that even with an ASIC architecture, the bootstrapping operation is quite slow. To accelerate SHE schemes, three different hardware accelerators [22], [20], [9] have appeared at CHES 2015. The accelerators use ring-LWE based SHE schemes and outperform their software counterparts by orders of magnitude; but still they take a reasonably large amount of time to evaluate functions of small complexity. The main problem is that, to support even a small multiplicative depth such as 9, the polynomial ring needs to have a degree 16,384 and a modulus size of 512 bit [20]. This makes the polynomial arithmetic costly. The size of the polynomial ring increases rapidly with the complexity of the function.

In this report we implement the recryption box on a Xilinx Virtex 6 FPGA board ML605. The board comes with a powerful FPGA and a high speed Gigabit Ethernet communication interface. The recryption box is connected to the cloud computer over the internet using the Ethernet interface. During a recryption operation, the cloud computer sends noisy (masked) encrypted data to the recryption box, which then returns refreshed encrypted data. To know the effect of the recryption box model on the run time, we have implemented *encrypted search* as the target application. In an encrypted search, clients send encrypted queries to the search engine, and the search engine returns encrypted results. Neither the search engine, nor the other parties come to know about the client's search queries. We show that with the usage of the recryption boxes we could reduce the encrypted search time by an order of magnitude.

The report is organized as follows. Section II provides the basic mathematical background of the LWE and ring-LWE problems, describes the FV somewhat homomorphic encryption scheme and the basic arithmetic operations. The next section describes the recryption box and its instantiations. In Section IV, an encrypted search algorithm is designed to benefit from the recryption box. Section V gives implementation details of the recryption box and describes the optimization techniques. Implementation results are provided in Section VI. The final section draws conclusions and discusses the possible future works in this direction.

II. Background

In this section we present a brief mathematical overview of the learning with errors (LWE) problem, its ring version ring-LWE, and the FV somewhat homomorphic encryption scheme.

A. The LWE and ring-LWE Problem

The LWE problem was introduced by Regev [21] in 2005. Its hardness can be reduced to the hardness of classical problems on lattices and can be parametrized by the dimension n of the lattice, an integer modulus q and an error distribution, which is typically taken as a discrete Gaussian distribution $\mathcal{X}$ over $\mathbb{Z}$.

The LWE problem is defined as follows. A secret vector $\mathbf{s}$ of dimension n is chosen uniformly in $\mathbb{Z}_q^n$. Then samples are produced by selecting uniform random vectors $\mathbf{a}_i$ and error terms e_i from the error distribution $\mathcal{X}$ and by computing $b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i \in \mathbb{Z}_q$. The LWE distribution $A_{s,\mathcal{X}}$ over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ is defined as the set of tuples $(\mathbf{a}_i, b_i)$. In the *decision LWE problem* the solver tries to distinguish with non-negligible advantage between the samples drawn from $A_{s,\mathcal{X}}$ and the same number of samples drawn uniformly from $\mathbb{Z}_q^n \times \mathbb{Z}_q$. In the *search LWE problem* the solver tries to compute $\mathbf{s}$. The number of samples is restricted to a polynomial in n for both the decision and search versions of the LWE problem.

The ring-LWE problem is a ring based version of the LWE problem and was introduced by Lyubashevsky, Peikert and Regev in [17]. Computations are performed in a polynomial ring $R_q = \mathbb{Z}_q[\mathbf{x}]/\langle f(x) \rangle$, where $f(x)$ is an irreducible polynomial of degree n. The ring-LWE problem is more efficient for constructing cryptosystems than the original LWE problem since the key size and computational complexity are no longer quadratic in n. The ring-LWE distribution on $R_q \times R_q$ consists of polynomial tuples $(a_i(x), b_i(x))$, where the coefficients of a_i are chosen uniformly from $\mathbb{Z}_q$ and $b_i(x)$ is computed as a polynomial $a_i(x) \cdot s(x) + e_i(x) \in R_q$. Here $s \in R_q$ is the secret polynomial, and e_i is the error polynomial sampled from an n-dimensional error distribution $\mathcal{X}$. In some cases, e.g: for 2^k-power cyclotomics, this error distribution can be taken as the product of n independent discrete Gaussians, but in general $\mathcal{X}$ is more complex. One can construct s by sampling the coefficients from $\mathcal{X}$ instead of sampling uniformly without any security implications [17]. An elegant public key encryption scheme was constructed in [17] based on the ring-LWE problem. The encryption scheme performs simple polynomial arithmetic such as polynomial multiplications, additions and subtractions, along with sampling from a discrete Gaussian distribution with a small parameter. Readers may follow [17] for detailed description of the encryption scheme, and [19], [24], [8] for the implementation techniques.

B. The FV Scheme

The FV somewhat homomorphic encryption scheme [11] works in the polynomial ring $R = \mathbb{Z}[x]/(f(x))$ with $f(x) = \Phi_d(x)$, the d-th cyclotomic polynomial of degree $n = \varphi(d)$. A plaintext is an element in the ring R_t for some small modulus t. Generally t is taken as 2. A ciphertext in this scheme consists of two elements in the ring R_q where q is the large modulus. The key generation and the encryption operations in the FV scheme require sampling from two probability distributions defined on R, namely χ_{key} and χ_{err} respectively. The security of the scheme is determined by the degree n of f, the size of q, and by the probability distributions. Following [17] one may sample the key and the error polynomials from a common distribution χ. Typically χ is a discrete Gaussian distribution with a small standard deviation. However in practice some authors take the key as a polynomial with coefficients from a narrow set like $\{-1, 0, 1\}$. In the following part of this section we explain some of the functions that are used to describe the FV algorithm.

$\texttt{WordDecomp}_{w,q}(a)$*:* This function is used to decompose a ring element $a \in R_q$ in base w by splicing each coefficients of a. For $l = \lceil \log_w(q) \rceil$, this function returns $a_i \in R$ with coefficients in $(-w/2, w/2]$, where $a = \sum_{i=0}^{l-1} a_i w^i$.

$\texttt{PowersOf}_{w,q}(a)$*:* This function scales an element $a \in R_q$ by the different powers of w. It is defined as $\texttt{PowersOf}_{w,q}(a) = (aw^i)_{i=0}^{l-1}$. The two functions can be used to perform a polynomial multiplication in R_q as

$$\langle \texttt{WordDecomp}_{w,q}(a), \texttt{PowersOf}_{w,q}(b) \rangle = a \cdot b \bmod q\,.$$

This expression has advantage in reducing the noise during homomorphic multiplications, as the first vector contains small elements (in base w).

The FV scheme uses an encryption scheme and three additional functions `Add`, `Mult`, and `ReLin` to perform function evaluations homomorphically on the encrypted data. In the following part of this section we describe the functions used in the FV scheme. For details of the functions, interested readers are referred to the original paper [11].

1) $\texttt{ParamsGen}(\lambda)$: For a given security parameter λ, choose a polynomial $\Phi_d(x)$, ciphertext modulus q and plaintext modulus t, and distributions χ_{err} and χ_{key}. Also choose the base w for $\texttt{WordDecomp}_{w,q}(\cdot)$. Return the system parameters $(\Phi_d(x), q, t, \chi_{err}, \chi_{key}, w)$. Following [11] we use a uniform signed binary distribution for χ_{key}. Additionally we set the plaintext modulus $t = 2$.
2) $\texttt{KeyGen}(\Phi_d(x), q, t, \chi_{err}, \chi_{key}, w)$: Sample polynomial s from χ_{key}, sample $a \leftarrow R_q$ uniformly at random, and sample $e \leftarrow \chi_{err}$. Compute $b = [-(as+e)]_q$. The public key consists of two polynomials $pk = \{b, a\}$ and the secret key is $sk = s$. The scheme uses another key called *relinearisation key* or $\mathbf{rlk}$ in the function `ReLin`. This key is computed as follows: first sample $\mathbf{a} \leftarrow R_q^l$ uniformly, then sample $\mathbf{e} \leftarrow \chi_{err}^l$, and then compute $\mathbf{rlk} = \{\mathbf{rlk}_0, \mathbf{rlk}_1\} = \{[\texttt{PowersOf}_{w,q}(s^2) - (\mathbf{e} + \mathbf{a} \cdot s)]_q, \mathbf{a}\} \in \{R_q^l, R_q^l\}$.
3) $\texttt{Encrypt}(pk, m)$: First encode the input message $m \in R_t$ into a polynomial $\Delta m \in R_q$ with $\Delta = \lfloor q/t \rfloor$. Next sample the error polynomials $e_1, e_2 \leftarrow \chi_{err}$, sample u uniformly from the signed binary distribution, and, compute the two polynomials $c_0 = [\Delta m + bu + e_1]_q \in R_q$ and $c_1 = [au + e_2]_q \in R_q$. The ciphertext is the pair of polynomials $\mathbf{c} = \{c_0, c_1\}$.
4) $\texttt{Decrypt}(sk, c)$: First compute a polynomial $\tilde{m} = [c_0 + sc_1]_q$. When $t = 2$, recover the plaintext message m by a decoding the coefficients of $\tilde{m}$. This decoding operation checks if the coefficient is in $(q/4, 3q/4)$ for a 1 bit and a 0 bit otherwise.
5) $\texttt{Add}(\mathbf{c_1}, \mathbf{c_2})$: For two ciphertexts $\mathbf{c_0} = \{c_{0,0}, c_{0,1}\}$ and $\mathbf{c_1} = \{c_{1,0}, c_{1,1}\}$, return $\mathbf{c} = \{c_{0,0} + c_{1,0}, c_{1,0} + c_{1,1}\}$.
6) $\texttt{Mult}(\mathbf{c}_1, \mathbf{c}_2, \mathbf{rlk})$: Compute $\tilde{\mathbf{c}}_{mult} = \{c_0, c_1, c_2\}$ where $c_0 = [\lfloor \frac{t}{q} \cdot c_{1,0} \cdot c_{2,0} \rceil]_q$, $c_1 = [\lfloor \frac{t}{q} \cdot (c_{1,0} \cdot c_{2,1} + c_{1,1} \cdot c_{2,0}) \rceil]_q$, and $c_2 = [\lfloor \frac{t}{q} \cdot c_{1,1} \cdot c_{2,1} \rceil]_q$. Next call the function $\texttt{ReLin}(\tilde{\mathbf{c}}_{\mathbf{mult}}, \mathbf{rlk})$.
7) $\texttt{ReLin}(\tilde{\mathbf{c}}_{\mathbf{mult}}, \mathbf{rlk})$: Compute a relinearised ciphertext $\mathbf{c}' = \{[c_0 + \langle \texttt{WordDecomp}_{w,q}(c_2), \mathbf{rlk}_0 \rangle]_q, [c_1 + \langle \texttt{WordDecomp}_{w,q}(c_2), \mathbf{rlk}_1 \rangle]_q\}$.

In our applications, we encrypt only one bit in one ciphertext. The encrypted bit remains in the least significant coefficient of the ciphertext polynomial. Note that in this setting, we encode only one bit, and not a polynomial, during an encryption. During a decryption we decode the least significant coefficient of $\tilde{m}$.

C. Basic Arithmetic Operations

In this report we implement the recryption box that performs only the decryptions and encryptions. From Section II-B we see that the main operations in the FV encryption and decryption are polynomial addition, multiplication and discrete Gaussian sampling. For multiplications of large polynomials, we use the number theoretic transform (NTT) algorithm due to its low time complexity [2]. For the discrete Gaussian sampling, we use the Knuth-Yao random walk method [15]. In the next subsections, we briefly describe the NTT based polynomial multiplication algorithm and the Knuth-Yao discrete Gaussian sampling algorithm.

The Number Theoretic Transform: or NTT is a Fast Fourier Transform (FTT) where the roots of unity are from a finite ring $\mathbb{Z}_q$. Since the roots of the unity are in $\mathbb{Z}_q$, all computations are performed on integers. In an n-point NTT with n a power of two, the input polynomial $a(x) = \sum_{j=0}^{n-1} a_j x^j \in \mathbb{Z}[x]$ is evaluated in the n points $x = \omega_n^i$, where $i = 0, \ldots, n-1$ and ω_n is a primitive n-th root of unity in the ring. The result of the polynomial evaluation is $\texttt{NTT}([a_j], \omega_n) = [a(\omega_n^0), a(\omega_n^1), \ldots, a(\omega_n^{n-1})]$

Input: Polynomial $a(x) \in \mathbb{Z}_q[\mathbf{x}]$ of degree $n-1$ and n-th primitive root $\omega_n \in \mathbb{Z}_q$ of unity
Output: Polynomial $A(x) \in \mathbb{Z}_q[\mathbf{x}]$ = NTT(a)

```
1  begin
2      A ← BitReverse(a);
3      for m = 2 to n by m = 2m do
4          ω_m ← ω_n^{n/m} ;
5          ω ← 1 ;
6          for j = 0 to m/2 − 1 do
7              for k = 0 to n − 1 by m do
8                  t ← ω · A[k + j + m/2] ;
9                  u ← A[k + j] ;
10                 A[k + j] ← u + t ;
11                 A[k + j + m/2] ← u − t ;
12             ω ← ω · ω_m ;
13 end
```

Algorithm 1: *Iterative NTT Algorithm from [5]*

. A straightforward evaluation of the input polynomial in the n-points will result in a quadratic complexity algorithm. The evaluation is done in $\theta(n \log n)$ time by utilizing a special property of the primitive root of unity in the NTT algorithm. An NTT can be computed in both recursive and iterative manner. For efficient implementation, the iterative version is more popular [19], [24], [8]. In Algorithm 1, we have shown an in-place iterative version of the NTT algorithm [5].

In line 2 the coefficients of the input polynomial a are rearranged using the function call $\texttt{BitReverse}(a)$. Next, three nested loops with loop variables m, j, and k compute new coefficients from the old coefficients. In the outer j-loop, powers of the primitive roots are assigned to a variable ω. The variable is known as the *twiddle factor*. In the innermost loop, two coefficients are processed at a time. This part of the algorithm is known as the *butterfly operation*. New values of the twiddle factor are computed in the loop with the variable j. More details of the algorithm are available in [5]. The inverse transform is known as the inverse NTT. It can be computed simply as $\frac{1}{n}\texttt{NTT}(\cdot, \omega_n^{-1})$.

Multiplication of two polynomials in the NTT domain is simply a coefficient wise multiplication of the two polynomials. Hence use of NTT leads to a fast polynomial multiplication algorithm. The two input polynomials are first mapped into NTT domain. Then the two polynomials are multiplied coefficient wise. In the end, an inverse NTT is performed to get back the result of the polynomial multiplication. The (reduced) multiplication of two polynomials a and b can be computed easily in the ring $S_q = \mathbb{Z}_q[x]/(x^n - 1)$ as follows:

$$c = \texttt{NTT}_{\omega_n}^{-1}\big(\texttt{NTT}_{\omega_n}(a) * \texttt{NTT}_{\omega_n}(b)\big) \tag{1}$$

Here $*$ denotes coefficient-wise multiplication.

Efficient Multiplication in R_q: If we consider a polynomial multiplication in $R_q = \mathbb{Z}_q[\mathbf{x}]/\langle f(x)\rangle$ where $\langle f(x)\rangle$ is an arbitrary reduction polynomial of degree n, then the first step is a normal polynomial multiplication over $\mathbb{Z}_q$. The result of this multiplication is a polynomial of degree $2n-2$. Next the result is reduced modulo $\langle f(x)\rangle$ to get the modular reduced result in R_q. Since the result of the normal polynomial multiplication over $\mathbb{Z}_q$ has a degree $2n-2$, the multiplication requires computation of $2n$ point NTTs [5].

One can do much better if the polynomial ring R_q possesses a special structure. For example, if the irreducible polynomial is $f(x) = x^n + 1$ and $n = 2^k$, then one can use a technique known as the *negative wrapped convolution* to get the $f(x)$-reduced result directly. Moreover, with the negative wrapped convolution technique, one needs to compute only n point NTTs instead of $2n$ point NTTs. There are some minor overheads due to the precomputations and postcomputations that are required for the negative wrapped convolution [24].

Discrete Gaussian Sampling using the Knuth-Yao Random Walk: In the FV encryption operation, the noise polynomials e_1 and e_2 are constructed from a discrete Gaussian distribution χ_{err}. For our implementation, the parameter of the discrete Gaussian distribution is small. In [15] Knuth and Yao proposed a random walk algorithm to sample from a constant discrete distribution. This algorithm stores the probabilities of the sample points in binary representation, and then generates a binary tree on-the-fly from the bits of the probabilities. The binary tree is called the Discrete Distribution Generating (DDG) tree. The DDG tree consists of intermediate nodes and terminal nodes. The terminal nodes lie on the right hand side of any level of the DDG tree and contain the sample points. During a sampling operation, a random walk starts from the root of the tree and uses a random bit to jump to the next level of the tree. When the random walk hits a terminal node of the DDG tree, the algorithm stops and the value of the terminal node is output as the result of the sampling process. Detailed descriptions about the implementations techniques can be found in [25]. The Knuth-Yao random walk algorithm is very efficient for implementing compact and fast discrete Gaussian sampler architecture when the parameter of the distribution is small.

III. Instantiations of the Recryption Box

In this section, we describe two possible instantiations of the recryption box and analyze their security and ease of use. In all cases, the parameters of the scheme become independent of the minimum size required for bootstrapping, resulting in faster homomorphic evaluations overall. We also consider possible applications of the setup described in this report.

Most homomorphic encryption schemes (including FV) admit an operation called key switching. Key switching allows to transform a ciphertext encrypted under one public key, into a valid ciphertext encrypted under a different public key. More in detail: assume the recryption box has its own private/public key pair (s_r, b_r, a_r) and the i-th user's keypair is (s_i, b_i, a_i). The user can then compute the key switching key as follows: he samples a vector $\mathbf{u}$ of l elements uniformly from χ_{key} (in our case a signed binary distribution), and two vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ of l elements from χ_{err}. Next he computes the key switching key $\{\mathbf{ksk}_{0_i}, \mathbf{ksk}_{1_i}\} = \{\mathtt{PowersOf}_{w,q}(s_i) + \mathbf{u} \cdot b_r + \mathbf{e}_1 \in R_q^l, \mathbf{u} \cdot a_r + \mathbf{e}_2 \in R_q^l\}$. The $\{\mathbf{ksk}_{0_i}, \mathbf{ksk}_{1_i}\}$ together with $\{b_i, a_i\}$ is sent to the cloud. The cloud uses the key switching key to switch a ciphertext $\{c_{0_i}, c_{1_i}\}$ encrypted under the user's public key to a valid ciphertext $\{c_{0_r}, c_{1_r}\}$ encrypted under the box's public key as follows: $\{c_{0_r}, c_{1_r}\} = \{\langle \mathtt{WordDecomp}_{w,q}(c_{1_i}), \mathbf{ksk}_{0_i}\rangle + c_0, \langle \mathtt{WordDecomp}_{w,q}(c_{1_i}), \mathbf{ksk}_{1_i}\rangle$. Before sending the ciphertext $\{c_{0_r}, c_{1_r}\}$ to the box, the cloud additively masks it (using the fact that the scheme is additively homomorphic) to obtain $\{c'_{0_r}, c'_{1_r}\}$. The ciphertext $\{c'_{0_r}, c'_{1_r}\}$ together with user's public key $\{b_i, a_i\}$ is sent to the box, who decrypts it using its own private key and freshly encrypts it using the user's public key. The resulting ciphertext $\tilde{c}_i$ is then sent back to the cloud, who removes the additive mask it added before.

For the above setup to offer any security at all, the following assumptions have to be made: firstly, we assume that both the box and cloud are honest but curious. In particular, the cloud has to apply a random mask before sending the ciphertext to the box such that it cannot recover the underlying plaintext. And in turn, the box has to execute the encryption correctly by choosing random error polynomials. Secondly, we assume that the cloud and box do not collude, e.g. the key switching key $\{\mathbf{ksk}_{0_i}, \mathbf{ksk}_{1_i}\}$ should not be given to the box since it would allow the box to derive the private key of the user. The advantage of this setup clearly is that a single recryption box can deal with many users. The downsides are the slightly stronger security assumptions and the extra operations involved such as key switching and additive masking by the cloud.

The second instantiation does not rely on a key switching key and makes it much more difficult for the cloud and the recryption box to collude by using a threshold scheme to split the secret key over several parties and using a distributed decryption protocol. The secret key can be split using a th out of n Shamir threshold sharing over the ring R_q. The i-th party receives a share $s_{r_i} \in R_q$ that equals the evaluation of a random polynomial $p(x)$ of degree $th-1$ in a public value $a_i \in R_q$ assigned to each party (one could sometimes even take $a_i = i$), i.e. $s_{r_i} = p(a_i)$. The secret key s of the user can then be obtained as $s = p(0)$, and it is clear that any set of th valid shares allows to recover s using for instance Lagrange interpolation. Denote the i-th Lagrange multiplier (for the set of th contributors) by $\lambda_i = \prod_{j \neq i} a_j/(a_j - a_i)$, then s can be recovered as $s = \sum_i \lambda_i s_{r_i}$. Denote by $\tilde{s}_{r_i}$ the scaled share $\lambda_i s_{r_i}$, then s can be simply recovered as $s = \sum_i \tilde{s}_{r_i}$. The main advantage of using Shamir secret sharing is that it defines a ring homorphism between $R_q, +, \cdot$ and $R_q^{th}, +, \cdot$. In particular, any algebraic expression in R_q can be recovered from executing the same expression on each of the th shares and reconstructing the result using interpolation.

The distributed decryption protocol will work in two steps. In the first step, for a given ciphertext (c_0, c_1) each party computes $d_i = \tilde{s}_{r_i} \cdot c_1 + e_i$ where e_i is a Gaussian distributed error polynomial. Note that recovering $\tilde{s}_{r_i}$ from d_i is hard since this corresponds precisely to the ring-LWE problem, so one party cannot recover another party's share. The shares d_i are then distributed over an encrypted channel to the other parties. In the second step, each party adds the shares to recover $c_1 \cdot s + e$. Now the end-party (or any party) then recovers the message m as $m = \lfloor \frac{t}{q} \cdot [c_0 + s \cdot c_1 + e]_q \rceil \in R_t$ and returns a fresh encryption of m as the final result. Since the end-party recovers the message m, it is required that the server additively masks the message before sending it to the recryption boxes. The security of the system now relies on the fact that not more than $th-1$ parties collude with the cloud server and that the cloud server uses additive masking.

IV. Encrypted search

To know the effect of the proposed recryption box in real life, we have implemented encrypted search as the target application. The reason behind this particular choice is mainly because of its future prospects. In an encrypted search, a client sends an encrypted keyword to the search engine and then the search engine returns the search-response in an encrypted format to the client. Due to its *encrypted* nature, the search engine remains oblivious of the search keyword. Such an encrypted search application is possible when the encryption scheme is homomorphic.

In the search engine, the search results are stored in unencrypted format in a table (as table entries) indexed by the numeric representations of the search keywords. During an encrypted search, a client's encrypted keyword is compared with the encryptions of the table indexes, and then the comparison results are used to perform arithmetic on the encryptions of the table entries. Since the search table is in unencrypted format, the search engine needs to encrypt the entire table under the public key of the client in order to perform homomorphic operations. However in reality the search engine needs to encrypt only two bits instead of the entire table. The client sends her public key along with the encrypted keyword to the search engine. Next, the search engine encrypts bit-0 and bit-1 using the public key of the client and constructs encryptions of the search table indexes and the entries by simply replacing the plaintext bits with the encryptions of bit-0 and bit-1. After the completion of the search operation, the result is an encryption of the search result that is associated with the search keyword. The total amount of data exchange between the client and the search engine is: the public key of the user, the homomorphic encryption of the search keyword, and the homomorphic encryption of the search result.

Input: Encrypted search keyword $K = \{k_{l-1} \dots k_0\}$ and the client's public key pk
Output: Encrypted search result $R = \{r_{p-1} \dots r_0\}$

1 **begin**
2 $e_0 \leftarrow enc(0, pk)$;
3 $e_1 \leftarrow enc(1, pk)$;
4 $R \leftarrow \{e_0, \dots, e_0\}$;
5 **for** $index = 0$ *to* $2^l - 1$ **do**
6 $C \leftarrow \{e_{\overline{index}_{l-1}}, \dots, e_{\overline{index}_0}\}$; /* enc. of $\overline{index}$ */
7 $T \leftarrow \{add(k_{l-1}, c_{l-1}), \dots, add(k_0, c_0)\}$;
8 $b \leftarrow t_0$;
9 **for** $i = 1$ *to* $l - 1$ **do**
10 $b \leftarrow mul(b, t_i)$;
11 $D \leftarrow table[index]$;
12 **for** $i = 0$ *to* $p - 1$ **do**
13 **if** $d_i = 1$ **then**
14 $r_i \leftarrow add(r_i, b)$; /* i-th bit of R */
15 **end**

Algorithm 2: *Linear encrypted search*

Input: w-bit chunk of encrypted keyword $K = \{k_{w-1} \dots k_0\}$ and client's public key pk
Output: Precomputation table $precomp[]$ with 2^w entries

1 **begin**
2 $e_0 \leftarrow enc(0, pk)$;
3 $e_1 \leftarrow enc(1, pk)$;
4 **for** $index = 0$ *to* $2^w - 1$ **do**
5 $C \leftarrow \{e_{\overline{index}_{w-1}}, \dots, e_{\overline{index}_0}\}$;
6 $T \leftarrow \{add(k_{w-1}, c_{w-1}), \dots, add(k_0, c_0)\}$;
7 $b \leftarrow t_0$;
8 **for** $i = 1$ *to* $w - 1$ **do**
9 $b \leftarrow mul(b, t_i)$;
10 $precomp[index] \leftarrow b$;
11 **end**

Algorithm 3: *Precomputation in linear search*

There are several algorithms to search in a plaintext database. The most efficient ones such as binary search or half-interval search [27] have logarithmic complexity. But in the case of an encrypted search, the search algorithm has linear complexity as it does not see the search term in plaintext and thus has to go through all of the database entries. A linear encrypted search operation is shown in Algorithm 2. The table to be searched is represented as an array of 2^l elements and the elements are accessed using the l-bit *index* variable. We assume that the table entries (i.e. the search data in any location) are of p bit. The encrypted search keyword K is a string of l ciphertexts k_i. In the start phase, the algorithm computes the encryptions e_0 and e_1 of bit-0 and bit-1 respectively (line 2 and 3), and then initializes the accumulator R to a string of p encryptions of bit-0 (line 4). Next, in the search phase the *for*-loop goes through all the indexes of the table. In line 6 the algorithm constructs an encryption of the one's complement of *index* (i.e. $\overline{index}$) in the variable C by concatenating the encryptions of the bits of $\overline{index}$. Next the algorithm adds the encrypted one's complement of the index with the encrypted keyword and obtains T, and then cumulatively multiplies the l encrypted bits of T to get a single encrypted bit b (lines 8-10). Note that b will be an encryption of bit-1 only for the loop $index$ that is equal to the unencrypted search keyword; otherwise b will be an encryption of bit-0. Now the algorithm fetches the table entry and stores it in the variable D. In the next part of the loop, the algorithm adds b for each nonzero bits d_i of D with the encrypted bits r_i of the accumulator R. Note that only when the unencrypted search keyword matches with the index of the search table, this b is an encryption of bit-1, and hence an encryption of the associated search result is added with the R; for all other indexes, encryptions of zeros are added with the accumulator. In this way when the for-loop covers all the indexes of the search table, we get an encryption of the search result in R.

Faster linear search on encrypted data

The most costly part in Algorithm 2 is the homomorphic multiplication of the l encrypted bits of T in lines 8 to 10. We reduce the number of homomorphic multiplications with the help of a window based pre-computation technique shown in Algorithm 3. With a window size w, the given encrypted keyword is split into l/w chunks. For simplicity let us assume that l is a multiple of w. Then for each chunk of the encrypted keyword, a pre-computation table is constructed. The algorithm considers all 2^w w-bit indexes: first an encryption of the one's complement of the index is added to the keyword-chunk to obtain T, and then the w encrypted bits in T are multiplied together to obtain the ciphertext b. Next b is stored in the precomputation table.

The search algorithm 4 uses the pre-computation tables to speedup computation. The loop *index* is split into l/w chunks in lines 4-5, and then the pre-computation tables corresponding to the chunks are accessed in line 6 to perform the cumulative multiplications in line 9. In comparison to Algorithm 2 this algorithm reduces the number of homomorphic multiplications

Input: Encrypted search keyword $K = \{k_{l-1} \dots k_0\}$ and the client's public key pk
Output: Encrypted search result $R = \{r_{p-1} \dots r_0\}$

begin
  $R \leftarrow \{e_0, \dots, e_0\}$;
  for $index = 0$ *to* $2^l - 1$ **do**
    for $i = 0$ *to* $l/w - 1$ **do**
      $chunk_i \leftarrow \{\overline{index}_{iw+w-1} \dots \overline{index}_{iw}\}$;
    $b \leftarrow precomp_0[chunk_0]$;
    for $i = 1$ *to* $l/w - 1$ **do**
      $temp \leftarrow precomp_i[chunk_i]$;
      $b \leftarrow mul(b, temp)$;
    $D \leftarrow table[index]$;
    for $i = 0$ *to* $p - 1$ **do**
      if $d_i = 1$ **then**
        $r_i \leftarrow add(r_i, b)$;
end

Algorithm 4: *Precomputation assisted linear encrypted search*

within the search loop by a factor w. The new algorithm requires additional memory to store the $2^w \frac{l}{w}$ precomputed table entries.

V. Implementation

In this report we implement a fast architecture for the proposed recryption box and then use this box to assist an encrypted search engine running on a server machine. The search algorithm is written in high level C and the recryptio box is implemented as a hardware module running on Xilinx ML605 board. The search engine performs homomorphic evaluations on the encrypted data. When the number of homomorphic evaluations reaches the maximum depth supported by the parameter set of the homomorphic encryption scheme, the search engine blinds the encrypted data and then sends it to the recryption box over a Gigabit Ethernet channel. After a recryption, fresh encrypted data is sent back to the search engine.

A. Parameter set used in the implementation

We use the FV scheme [11] as the homomorphic encryption scheme. We target fastest computation time and at least 90-bit security. Since the computation time of the FV homomorphic encryption scheme has almost a quadratic-complexity with respect to the multiplicative depth, we chose a parameter set that supports the minimum multiplicative depth, i.e. the depth one. Following [16] we set the dimension of the polynomial ring R_q to $n = 1024$, the modulus q to a 40-bit integer, and the parameter s of the discrete Gaussian distribution to 11.32. This parameter set has 96-bit security [1].

B. Algorithmic optimizations for efficient architecture

The basic computations in the FV encryption and decryption are discrete Gaussian sampling, polynomial addition and multiplication, and decoding-encoding. Among the arithmetic operations, polynomial multiplication is the costliest one. We use the NTT algorithm 1 to perform polynomial multiplication in the most efficient way. For the chosen parameter set, integer arithmetic operations are performed with respect to a 40-bit modulus q. To achieve faster processing through parallelization, we use the *Chinese Remainder Theorem* (CRT) to split 40-bit arithmetic into two parallel 20-bit arithmetic operations. We take the modulus q as a product of two 20-bit primes $q_0 = 878593$ and $q_1 = 890881$. Since both q_0 and q_1 are congruent to 1 modulo $2n$, we use the negative-wrapped convolution for faster NTT computation. With the application of CRT each operation modulo q during a polynomial arithmetic turns into two parallel 20-bit operations modulo q_0 and q_1. Note that the security of the encryption scheme does not get affected by this choice for q.

This parallel nature of the algorithm is very useful since the underlying hardware platform is also parallel. Hence two computation threads modulo q_0 and q_1 run in parallel. Beside this parallel processing, there is an another advantage of splitting the computation into two half-sized integers. Xilinx Virtex 6 FPGAs have fast but small 25×18 DSP multipliers. Hence a 20-bit coefficient can be easily processed by the small DSP multipliers (with some additional logic elements). Moreover smaller integer size is also very helpful to keep a pair of coefficients of a residue polynomial in one BRAM address and reduce the memory access overhead by using Algorithm 2 of [24] for the *memory efficient NTT* computation (shown in the Appendix). We need only one 36K BRAM slice to store a residue polynomial with two coefficients in one address.

Though CRT allows parallel processing, it has the overhead of inverse-CRT computation whenever the computation demands arithmetic in modulo q. For the proposed recryption box, inverse-CRT is required only during the decoding phase of the FV decryption operation. This is because the decoding operation needs to compare the coefficients with $q/4$ and $3q/4$. However, for our application this inverse-CRT computation is actually not a major overhead since we only need to decode the least significant coefficient of the ciphertext polynomial, and it is known that the remaining coefficients decode to zeros. This is because of the fact that we encrypt only one bit in a ciphertext using the FV homomorphic scheme, and the encrypted bit remains in the least significant coefficient of the ciphertext. Hence we compute the inverse CRT only once. Note that all of the remaining arithmetic operations in the FV encryption and decryption can be performed on the CRT-represented shares.

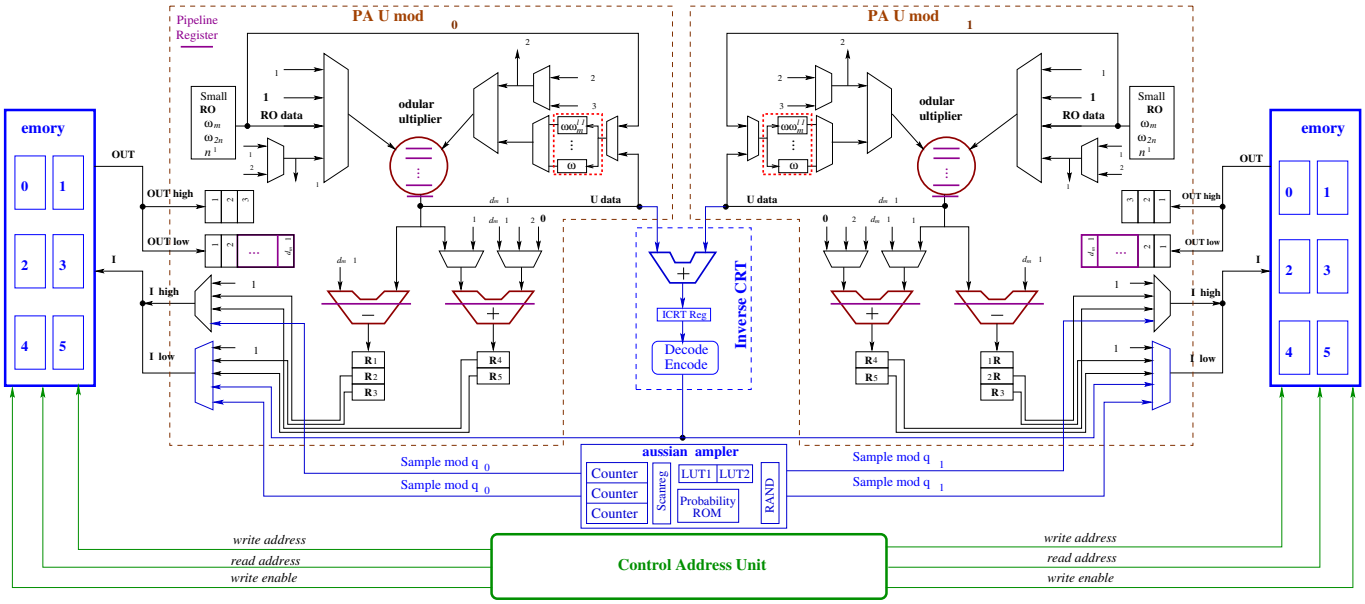

Fig. 1. Architecture of The Recryption Box

C. Architecture

The internal architecture of the processor part of the recryption box is shown in Fig. 1. The processor has two symmetric polynomial arithmetic and logic units (PALUs) for performing residue arithmetic modulo q_0 and q_1 in parallel. The PALUs are designed following the footprint of the compact ring-LWE encryption architecture proposed in [24]. Each PALU is connected to a memory file that keeps the residue polynomials.

1) PALUs: Each PALU has modular multiplier, modular addition and subtraction circuits to perform arithmetic operations on the input coefficients during a polynomial operation. The integer multipliers inside the modular multipliers are implemented using DSP multipliers. For the modular reduction of the integer multiplication result, we have used window based modular reduction technique. This modular reduction technique does not depend on the choice of the 20-bit prime modulus and is thus generic. The critical path of the PALU is through the modular multiplier and then through the addition (or subtraction) circuit. We split the critical path in almost equal delay sections using pipelines and achieve high operating frequency.

During an NTT computation on a residue polynomial, the control logic follows the *memory efficient NTT* (Algorithm 5 in Appendix) and processes two coefficient-pairs $(A[k+j+m/2], A[k+j])$ and $(A[k+m+j+m/2], A[k+m+j])$ (i.e. four coefficients) simultaneously excluding the last loop.

In this architecture we do not store the fixed twiddle factors, and instead compute them on the fly during an NTT operation. The small ω-ROM stores the $\log(n)$ twiddle factors ω_m (line 4 in algorithm 1) to generate the new twiddle factors (line 12). The modular multiplier circuit is reused for computing coefficient wise multiplications, and the modular addition/subtraction circuit is reused for coefficient wise addition/subtraction operations during polynomial arithmetic.

D. Inverse CRT

The inverse CRT combines two residues and computes the coefficient modulo q. Let a_0 and a_1 be the two residues. Then the equation for the inverse CRT computation is shown below.

$$a = [a_0 \cdot q_1^{-1}]_{q_0} \cdot q_1 + [a_1 \cdot q_0^{-1}]_{q_1} \cdot q_0 \mod q \tag{2}$$

In the above computation $[q_1^{-1}]_{q_0}$, q_1, $[q_0^{-1}]_{q_1}$, and q_0 are constants. Hence we store these constants in the PALUs. The PALU for the residue q_0 computes $[a_0 \cdot q_1^{-1}]_{q_0} \cdot q_1$ part of Eq. 2 and the other PALU computes the remaining part. The first PALU computes in two steps: first it computes $[a_0 \cdot q_1^{-1}]_{q_0}$ using the modular multiplier, then it multiplies this 20-bit result with q_1 to get the 40-bit integer multiplication result $[a_0 \cdot q_1^{-1}]_{q_0} \cdot q_1$. The other PALU associated with the modulus q_1 computes $[a_1 \cdot q_0^{-1}]_{q_1} \cdot q_0$ in a similar way. These two 40-bit outputs from the two PALUs are added together using the adder in the *Inverse CRT* block (Fig. 1). Next the 41-bit addition result is reduced by the 40-bit q by performing one subtraction. The *Decode-Encode* block in Fig. 1 compares the inverse CRT output with $q/4$ and $3q/4$ and then encodes the coefficient to either 0 or $q/2$. Note, that this inverse CRT is performed only once as we need to decode-encode only the least significant coefficient in the FV decryption.

E. The Memory

The memory of the recryption box consists of two independent memory files (Fig. 1) for the two PALUs. Each of the two memory files contains six RAM blocks M0, M1, M2, M3, M4 and M5, each containing 512 words. The coefficients of residue polynomials are kept as pairs in these RAM words. Each of these six RAM blocks consumes one 36K BRAM slice. During a recryption operation, the box's share of the client's secret key is loaded in M0, and the clients public key is loaded in M1 and M2. Since these keys are constants for a client, they are kept in the NTT domain to avoid unnecessary computation.

In the decryption phase, RAM blocks M3 and M4 are used to store the two polynomials c_0 and c_1 of the ciphertext. Hence a forward NTT of M3 is computed followed by a coefficient-wise multiplication of M0 and M3, and then the result is stored in M3. Next an inverse NTT is computed on $M3$ and this ends the polynomial multiplication in the multiparty decryption operation.

During the encryption phase, the encoded message is kept in M3. The noise polynomial e_1 is generated in M4, then added with the encoded message and finally the result is kept in M3. Another noise polynomial u is generated in M5. The two polynomial multiplications $pk_0 \cdot u$ and $pk_1 \cdot u$ are kept in M4 and M5.

F. The Discrete Gaussian Sampler

The DGS uses the Knuth-Yao algorithm [15] to generate the samples. The sampler is based on the hardware architectures proposed in [25], [24], [23] and designed for the discrete Gaussian distribution parameter 11.32. The precision and tail bounds of the designed sampler architecture satisfy a statistical distance of 2^{-90} from the accurate discrete Gaussian distribution. The probability bits of the sample points are stored in the *Probability ROM* in Fig. 1 in a column-wise manner. During a Knuth-Yao random walk, the probability bits are scanned bit-by-bit using a register *ScanReg*. The random walk needs random bits and the bits are supplied by a set of true random number generators (TRNGs). Since the speed of the sampling operation is critical for the encryption operation, the sampler architecture follows the lookup table method from [24] to reduce the number of cycles. This lookup table method uses the fact that the probability of hitting a sample point is large in the beginning of a Knuth-Tao random walk. Hence, the lookup tables store the outcomes of the random walks for all possible strings of random bits of small length. During the sampling operation, a string of random bits is used to address the lookup tables. With large probability, the lookup operation outputs a sample point. If the lookup operation fails to output a sample point, then the slow bit-by-bit scanning of the probability bits from the *Probability ROM* is performed. We use two lookup tables following [24] and the failure probability of a lookup operation is only 0.0016. We keep a set of nine parallel TRNGs to supply the random bits.

G. The Ethernet Communication Unit

The Xilinx ML605 development board has a single physical networking interface which is wired to the FPGA. This FPGA has four Embedded Tri-Mode Ethernet MAC cores [29] which are present in the silicon of the FPGA (hard-cores). To incorporate such a core in the design, the Xilinx CORE Generator is used to provide wrapper files which help to configure and interface the Ethernet MAC. Because high throughput is required and all the wiring between the physical interface (PHY) and the MAC on the FPGA are present, the Gigabit media-independent Inteface (GMII) is used.

Using the wrapper files (as generated by the Xilinx IPCore tool) to interface the hard-core MAC provides an easy-to-use interface, consisting of four signals: the data vector (8-bit wide), a data-valid signal, a data-user, and the data-last signal. For a more detailed explanation on these signals and their usage, the reader is referred to the Xilinx documentation [29], [28].

VI. Results

We have implemented the encrypted search (Algorithm 4) as the target application for performance evaluation. The search algorithm is a software program written using high-level C with GMP library for long integer arithmetic, and runs on a powerful server that has Intel(R) Xeon(R) CPU E5-2687W v3 with 40 cores running at 3.10GHz. During an encrypted search operation, the search engine, i.e. the server machine sends noisy ciphertexts to the third party recryption boxes over a gigabit Ethernet channel. The recryption boxes are implemented on Xilinx FPGA boards ML605. This board comes with a powerful Virtex 6 FPGA and a gigabit Ethernet for external communication [30].

In this implementation, we have restricted the size of the search table to 2^{16} entries. Thus the index of the search table has 16 bit width. We use an 8-bit window size in Algorithm 4. With this window size, the *index* variable in Algorithm 4 consists of two chunks, and hence only one homomorphic multiplication is required per iteration of the search loop. This is also very helpful in reducing the network traffic between the search engine and the recryption boxes, because the parameter set of the homomorphic scheme supports only one multiplication, and with only one multiplication per iteration of the search loop, the search engine does not need to refresh its encrypted data. The recryption boxes are required only during the precomputation phase of the encrypted search operation (Algorithm 3). Since this precomputation phase has a very small computation overhead with respect to the actual search loop, the network communication overhead is not a big issue.

The high level software implementation takes 21 seconds to perform one encrypted search operation. To know the actual effect of the proposed recryption box assisted encrypted search model, we have also implemented an encrypted search software

TABLE I
AREA OF THE RECRYPTION BOX ON XILINX VIRTEX-6 XC6VLX240T-1FF1156 FPGA

Component	Resource	Used	Avail.	Percentage
Recryption box	Slice Registers	2,684	301,440	0.9 %
	Slice LUTs	3,379	150,720	2.3 %
	BlockRAM36k	12	416	2.9 %
	DSP48	4	768	0.5 %
Processor	Slice Registers	1,848	301,440	0.6 %
	Slice LUTs	2,751	150,720	1.8 %
	BlockRAM36k	12	416	2.9 %
	DSP48	4	768	0.5 %

TABLE II
LATENCIES AND TIMINGS AT 125 MHZ

Operation	Clocks	Time
NTT	7181	0.0575 ms
INTT	9910	0.0793 ms
Coefficient wise add	1032	0.0083 ms
Coefficient wise mul	1040	0.0083 ms
Gaussian sampling	1080	0.0086 ms
Inverse CRT	28	0.0002 ms
Decryption	19191	0.153 ms
Encryption	34385	0.275 ms
Recryption †	53576	0.428 ms

† The recryption box is instantiated in the first mode (i.e. with key switching) or in the second mode (threshold sharing) with $th = 1$.

that does not use the recryption boxes. The parameter set for this implementation supports the full multiplicative depth of the encrypted search, and has polynomial dimension $n = 4096$, modulus size 141 bits, and Gaussian distribution parameter $s = 11.32$. The security analysis following [1] gives 96 bit security for this parameter set. The software takes 6 minutes and 40 seconds to perform an encrypted search on the same server machine, which is roughly 20 times slower than with the recryption box.

We have used mixed Verilog and VHDL to describe the recryption box architecture and have compiled the architecture using the Xilinx ISE 14.7 tool with a constraint file. The area requirements of the recryption box architectures are shown in Table I. The processor part of the recryption box consumes around 1.8% of the slice LUTs and 0.6% of the registers available in the FPGA. For the recryption box architectures, the additional area requirement is mostly due to the Ethernet wrapper and the small components that are used to perform the communication between the FPGA and the computer.

The latencies of different operations are shown in Table VI. In the design constraint file the operating frequencies of the clocks were set to 125MHz; both the Ethernet wrapper and the arithmetic unit run at 125 MHz, but using different clock domains. From the table we see that the most computation intensive operations are NTT and INTT. A decryption operation computes one NTT, one coefficient wise multiplication and one INTT, one coefficient wise addition, inverse CRT followed by a decode-encode of one coefficient. Thus it takes around 0.153 ms. An encryption computes additional discrete Gaussian sampling operations and thus takes slightly more amount of time that the decryption operation. One recryption operation is basically a decryption followed by an encryption and thus takes around 0.428 ms, excluding the cost of data transfer between the FPGA and the host computer. To know the overall time (data exchange + computation) of one recryption operation, we measured the actual timing from the FPGA board using a counter, and found this to be 0.6 ms.

To get a sense of comparison with the actual bootstrapping operation, we consider the FHE implementation on an Intel Core i7 processor running at 3.4GHZ in [6]. The authors in [6] implement the FHE scheme over integers and choose a very large parameter set to support the bootstrapping operation. One bootstrapping operation requires 172 s to refresh encrypted data. In comparison, using the proposed single-processor recryption box we can refresh a ciphertext in only 0.6 ms; this is roughly 2.8×10^5 times faster than the bootstrapping operation in [6]. Moreover the efficiency gain is not only restricted to the cleaning of the encrypted data. The large parameter set used in [6] also slows down the homomorphic multiplication and addition operations. One homomorphic multiplication in [6] takes 0.72 s; whereas for our parameter set it takes roughly 11 ms on a single core running at 3.1 GHZ.

VII. CONCLUSIONS AND FUTURE WORK

In this report we have proposed a recryption box model to assist fully homomorphic function evaluation. This recryption box is used to bypass the costly bootstrapping operation and achieve an order of magnitude speedup in homomorphic evaluation time. We described two possible instantiations of the recryption box and analyzed their security and ease of use. In our opinion, the main advantage of the recryption box is that the costly bootstrapping mechanism is no longer required, and therefore with this we can reduce the parameters of the somewhat homomorphic encryption scheme and achieve near practical evaluation

time. We demonstrated the soundness of our proposal by implementing a recryption box assisted encrypted search that achieves nearly twenty times speedup with respect to an implementation that does not use a recryption box.

We see several possibilities to improve the proposed work. In this work we designed the architecture for the recryption algorithm. In real life situations, the recryption box should have strong countermeasures against side channel and fault attacks. This is because the recryption box stores a share of the private key and performs partial decryptions using this share. Beside this, we could accelerate the encrypted search algorithm by designing dedicated hardware accelerators for the search engine. This will be specially helpful since the time for a homomorphic operation can be reduced by an order of magnitude using hardware accelerators.

VIII. Acknowledgements

This work was supported in part by the Research Council KU Leuven: C16/15/058. In addition, this work is supported in part by the Flemish Government, FWO G.0550.12N, G.00130.13N and FWO G.0876.14N, by the Hercules Foundation AKUL/11/19, and by the European Commission through the ICT programme under contract FP7-ICT-2013-10-SEP-210076296 PRACTICE, the Horizon 2020 research and innovation programme under contract No H2020-ICT-2014-644371 WITDOM and H2020-ICT-2014-644209 HEAT, and by the ERC Advanced Grant.

References

[1] M. R. Albrecht. Complexity estimates for solving lwe. https://bitbucket.org/malb/lwe-estimator/raw/HEAD/estimator.py.
[2] D. Bernstein. Fast Multiplication and its Applications. *Algorithmic Number Theory*, 44:325–384, 2008.
[3] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In M. Stam, editor, *Proceedings of the 14th IMA International Conference on Cryptography and Coding (IMACC 2013)*, volume 8308 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2013.
[4] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology — CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.
[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction To Algorithms*. http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/toc.htm.
[6] J.-S. Coron, T. Lepoint, and M. Tibouchi. Batch fully homomorphic encryption over the integers. Cryptology ePrint Archive, Report 2013/036, 2013. http://eprint.iacr.org/.
[7] J.-S. Coron, T. Lepoint, and M. Tibouchi. Scale-invariant fully homomorphic encryption over the integers. In H. Krawczyk, editor, *Public-Key Cryptography — PKC 2014*, volume 8383 of *Lecture Notes in Computer Science*, pages 311–328. Springer, 2014.
[8] R. de Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Efficient software implementation of ring-lwe encryption. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 339–344, 2015.
[9] Y. Doröz, E. Öztürk, E. Savas, and B. Sunar. Accelerating LTV based homomorphic encryption in reconfigurable hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 185–204, 2015.
[10] Y. Doröz, E. Öztürk, and B. Sunar. Accelerating fully homomorphic encryption in hardware. *IEEE Transactions on Computers*, 64(6):1509–1521, June 2015.
[11] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. http://eprint.iacr.org/.
[12] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing (STOC 2009)*, pages 169–178, 2009.
[13] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology — CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012.
[14] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology — CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013.
[15] D. E. Knuth and A. C. Yao. The Complexity of Non-Uniform Random Number Generation. *Algorithms and Complexity*, pages 357–428, 1976.
[16] T. Lepoint and M. Naehrig. A Comparison of the Homomorphic Encryption Schemes FV and YASHE. *IACR Cryptology ePrint Archive*, 2014:62, 2014.
[17] V. Lyubashevsky, C. Peikert, and O. Regev. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer Berlin Heidelberg, 2010.
[18] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW 2011)*, pages 113–124. ACM, 2011.
[19] T. Pöppelmann and T. Güneysu. Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware. In *Selected Areas in Cryptography – SAC 2013*, Lecture Notes in Computer Science, pages 68–85. Springer Berlin Heidelberg, 2014.
[20] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias. Accelerating homomorphic evaluation on reconfigurable hardware. Cryptology ePrint Archive, Report 2015/631, 2015. http://eprint.iacr.org/.
[21] O. Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM.
[22] S. S. Roy, K. Jarvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede. Modular hardware architecture for somewhat homomorphic function evaluation. Cryptology ePrint Archive, Report 2015/337, 2015. http://eprint.iacr.org/.
[23] S. S. Roy, O. Reparaz, F. Vercauteren, and I. Verbauwhede. Compact and side channel secure discrete gaussian sampling. Cryptology ePrint Archive, Report 2014/591, 2014. http://eprint.iacr.org/.
[24] S. Sinha Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact ring-lwe cryptoprocessor. In L. Batina and M. Robshaw, editors, *Cryptographic Hardware and Embedded Systems CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 371–391. Springer Berlin Heidelberg, 2014.
[25] S. Sinha Roy, F. Vercauteren, and I. Verbauwhede. High Precision Discrete Gaussian Sampling on FPGAs. In *Selected Areas in Cryptography – SAC 2013*, Lecture Notes in Computer Science, pages 383–401. Springer Berlin Heidelberg, 2014.
[26] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In H. Gilbert, editor, *Advances in Cryptology — EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010.
[27] L. F. Williams, Jr. A modification to the half-interval search (binary search) method. In *Proceedings of the 14th Annual Southeast Regional Conference*, ACM-SE 14, pages 95–101, New York, NY, USA, 1976. ACM.
[28] Xilinx. *LogiCORE IP Virtex-6 FPGA Embedded Tri-Mode Ethernet MAC wrapper (ug800)*, 3 2011. http://www.xilinx.com/support/documentation/ip_documentation/ug800_v6_emac.pdf.

[29] Xilinx. *Virtex-6 FPGA Embedded Tri-Mode Ethernet MAC (ug368)*, 3 2011. http://www.xilinx.com/support/documentation/user_guides/ug368.pdf.
[30] Xilinx. *ML605 Hardware User Guide*, 2012. http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf.

APPENDIX

Input: Polynomial $a(x) \in \mathbb{Z}_q[\mathbf{x}]$ of degree $n-1$ and n-th primitive root $\omega_n \in \mathbb{Z}_q$ of unity
Output: Polynomial $A(x) \in \mathbb{Z}_q[\mathbf{x}]$ = NTT(a)

begin
  $A \leftarrow BitReverse(a)$; /* Coefficients are stored in the memory as proper pairs */
  for $m = 2$ *to* $n/2$ *by* $m = 2m$ **do**
    $\omega_m \leftarrow m$-th primitiveroot(1) ;
    $\omega \leftarrow squareroot(\omega_m)$ or 1 /* Depending on forward or backward NTT */ ;
    for $j = 0$ *to* $m/2 - 1$ **do**
      for $k = 0$ *to* $n/2 - 1$ *by* m **do**
        $(t_1, u_1) \leftarrow (A[k + j + m/2], A[k + j])$ /* From MEMORY[k+j] */ ;
        $(t_2, u_2) \leftarrow (A[k + m + j + m/2], A[k + m + j])$ /* MEMORY[k+j+m/2] */ ;
        $t_1 \leftarrow \omega \cdot t_1$;
        $t_2 \leftarrow \omega \cdot t_2$;
        $(A[k + j + m/2], A[k + j]) \leftarrow (u_1 - t_1, u_1 + t_1)$;
        $(A[k + m + j + m/2], A[k + m + j]) \leftarrow (u_2 - t_2, u_2 + t_2)$;
        $MEMORY[k + j] \leftarrow (A[k + j + m], A[k + j])$;
        $MEMORY[k + j + m/2] \leftarrow (A[k + j + 3m/2], A[k + j + m/2])$;
      $\omega \leftarrow \omega \cdot \omega_n$;
  $m \leftarrow n$;
  $k \leftarrow 0$;
  $\omega \leftarrow squareroot(\omega_m)$ or 1 /* Depending on forward or backward NTT */ ;
  for $j = 0$ *to* $m/2 - 1$ **do**
    $(t_1, u_1) \leftarrow (A[j + m/2], A[j])$ /* From MEMORY[j] */ ;
    $t_1 \leftarrow \omega \cdot t_1$;
    $(A[j + m/2], A[j]) \leftarrow (u_1 - t_1, u_1 + t_1)$;
    $MEMORY[j] \leftarrow (A[j + m/2], A[j])$;
    $\omega \leftarrow \omega \cdot \omega_m$;
end

Algorithm 5: *Iterative NTT : Memory Efficient Version [24]*