Exploring the Feasibility of Fully Homomorphic Encryption

Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, *Senior Member*, *IEEE*, and Berk Sunar, *Member*, *IEEE*

Abstract—In 2010, Gentry and Halevi presented the first FHE implementation. FHE allows the evaluation of arbitrary functions directly on encrypted data on untrusted servers. However, even for the small setting with 2048 dimensions, the authors reported a performance of 1.8 s for a single bit encryption and 32 s for recryption on a high-end server. Much of the latency is due to computationally intensive multi-million-bit modular multiplications. In this paper, we introduce two optimizations coupled with a novel precomputation technique. In the first optimization called partial FFT, we adopt Strassen's FFT-based multiplication algorithm along with Barret reduction to speedup modular multiplications. For the encrypt primitive, we employ a window-based evaluation technique along with a modest degree of precomputation. In the full FFT optimization, we delay modular reductions and change the window algorithm, which allows us to carry out the bulk of computations in the frequency domain. We manage to eliminate all FFT conversion except the final inverse transformation drastically reducing the computation latency for all FHE primitives. We implemented the GH FHE scheme on two GPUs to further speedup the operations. Our experimental results with small parameter setting show speedups of 174, 7.6, and 13.5 times for encryption, decryption, and recryption, respectively, when compared to the Gentry–Halevi implementation. The speedup is enhanced in the medium setting. However, in the large setting, memory becomes the bottleneck and the speedup is somewhat diminished.

Index Terms—Fully homomorphic encryption, GPU, large-number multiplication, modular reduction

1 Introduction

In the past decade, one of the most significant advances in cryptography has been the introduction of the first fully homomorphic encryption scheme (FHE) by Gentry [1]. This advance not only solved an open problem posed by Rivest [2], but also opened the door to many new applications. Indeed, using a FHE one may perform an arbitrary number of computations directly on the encrypted data without revealing of the secret key. Thus an untrusted party, such as a remotely hosted server, may perform computations on behalf of the owner on the data without compromising privacy. This property of FHE is precisely what makes FHE invaluable for the cloud computing industry today. For instance, it was recognized early on in [1] that FHE is ideally suited to protect sensitive data on untrusted cloud servers. Considering the recent growth in the adoption of cloud services, it is foreseeable that FHE schemes will have a transforming effect on personal computing in the coming years.

Despite its promise, FHE is nowhere near ready for real-life deployment due to serious efficiency impediments. The first implementation of an FHE variant was proposed by Gentry and Halevi [3], who presented an impressive array of optimizations with the goals of reducing the size of the public-key and improving the performance of the primitives. Still, encryption of one bit takes more than a second on a high-end Intel Xeon based server, while recrypt primitive takes nearly half a minute for the lowest security setting. Furthermore, after every few bit-AND operations a recrypt operation must be applied to reduce the noise in the ciphertext to a manageable level. When application specific FHE hardware is considered, the situation becomes even worse. In [4], an FPGA implementation draft for improving the speed of FHE primitives was proposed. However, no implementation results were presented. Clearly, much work is needed before FHE becomes practically useful.

In this paper we take another step in this direction. We present a GPU acceleration of the FHE variant introduced by Gentry and Halevi [3]. Our implementation shows significant improvement in speed over the Gentry-Halevi CPU implementation. Since GPU based cloud computing services are already available, e.g. on Amazon's EC2 cluster GPU instances, our approach is well supported on existing cloud platforms. The GPU approach also makes sense when one considered the rapid progression in the processor industry. With continuous architectural improvements in recent years, GPUs have evolved into a massively parallel, multithreaded, many-core processor system with tremendous computational power. Owing to introduction of the Compute Unified Device Architecture (CUDA) programming paradigm, a vast of computation problems outside of the graphics domain have benefited from the superior performance of GPUs. Among the examples of the general purpose GPU (GPGPU) computing initiative are FFT [5], data processing [6], and many other science and engineering applications [7].

In this work, we present an array of algorithmic optimizations to the Gentry-Halevi FHE algorithm [3]. We achieve the best performance by reformulating the iterations to delay the modular reductions in the implementation of the encryption and recryption primitives to eliminate costly back and forth conversions normally experienced during Strassen's FFT

• *The authors are with the Worcester Polytechnic Institute, Worcester, MA 01609. E-mail: {weiwang, hhyybest, lchen, xhuang, sunar}@wpi.edu.*

Manuscript received 28 Dec. 2012; revised 12 June 2013; accepted 20 July 2013. Date of publication 01 Aug. 2013; date of current version 11 Feb. 2015. Recommended for acceptance by P. Schaumont.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.154

based integer multiplication algorithm. Thus, we are performing almost all of the computations in the FFT domain. Modular reductions are still required in the final step of the implementations. For this we utilize Barrett's modular reduction algorithm. We achieve further improvements by choosing a GPU platform that matches the computational needs of the Gentry-Halevi FHE. For the small parameter setting, we achieve a 13.5-fold speedup for the most critical primitive, i.e. recryption, and 174-fold and 7.6-fold speedup in the cases of encryption and decryption, respectively, on the NVIDIA GTX690 over the CPU implementation on Intel i7 3770 K at 3.5 GHz.

The rest of the paper is organized as follows: Section 2 briefly introduces the Gentry-Halevi FHE; Section 3 provides the design and performance of large number adder and subtraction algorithm; a large-number modular multiplier design is presented in Section 4; the complete GH-FHE GPU implementation is discussed in Section 5 which is followed by the experiment results in Section 6 and conclusions in Section 7.

2 Gentry's Fully Homomorphic Encryption

Informally a homomorphic encryption scheme refers to an encryption function that allows one to induce a binary operation on the plaintexts while only manipulating the ciphertexts without the knowledge of the encryption key: $E(x_1) \star E(x_2) = E(x_1 \otimes x_2)$. If the scheme supports the efficient homomorphic evaluation of any efficiently computable function, it is called a fully homomorphic encryption scheme (FHE). With FHE, an honest but curious party can perform any computation directly with encrypted result without gaining access to the plaintext.

The first FHE was proposed by Gentry in [1], [8]. The Gentry-Halevi FHE variant with a number of optimizations and the results of a reference implementation were presented in [3]. However, this preliminary implementation is far too inefficient to be used in any practical applications. Here we only present a high-level overview and refer the reader to the original work in [3] for details.

Encrypt: To encrypt a bit $b \in \{0, 1\}$ with a public key (d, r), Encrypt first generates a random "noise vector" $u = \langle u_0, u_1, \ldots, u_{n-1} \rangle$, with each entry chosen as 0 with some probability p and as ± 1 with probability $(1-p)/2$ each. Clearly, p will determine the hamming weight of the random noise u. Gentry showed in [3] that u can contain a large number of zeros without impact the security level, i.e., p could be very large.

Then the message bit b is encrypted by computing

$$c = [b + u(r)]_d = \left[b + 2 \sum_{i=1}^{n-1} u_i r^i \right]_d, \tag{1}$$

where d and r is part of the public key.

Eval: When encrypted, arithmetic operations can be performed directly on the ciphertext with corresponding modular operations. Suppose $c_1 = \mathsf{Encrypt}(m_1)$ and $c_2 = \mathsf{Encrypt}(m_2)$, we have:

$$\mathsf{Encrypt}(m_1 + m_2) = (c_1 + c_2) (\mathrm{mod}\ d),$$
$$\mathsf{Encrypt}(m_1 \cdot m_2) = (c_1 \cdot c_2) (\mathrm{mod}\ d).$$

Decrypt: An encrypted bit can be recovered from a ciphertext c by computing

$$m = [c \cdot w]_d \bmod 2, \tag{2}$$

where w is the private key and d is part of the public key.

Recrypt: The Recrypt process is implemented by homomorphically evaluating the decryption circuit on the ciphertext. However, due to the fact that we can only encrypt a single bit and that we can only evaluate a limited number of arithmetic operations, we need an extremely shallow decryption method. In [3], the authors discussed a practical way to re-organize the decryption process to make this possible. Informally, the private key is divided into s pieces that satisfy $\sum^s w_i = w$. Each w_i is further expressed as $w_i = x_i R^{l_i} \bmod d$ where R is constant, x_i is random and $l_i \in \{1, 2, \ldots, S\}$ is also random. The recryption process can then be expressed as:

$$\begin{aligned} m &= [c \cdot w]_d \bmod 2 \\ &= \left[\sum^S c x_i R^{l_i} \right]_d \bmod 2 \\ &= \left[\sum^S c x_i R^{l_i} \right]_2 - \left[\left\lfloor \left(\sum^S c x_i R^{l_i} \right) / d \right\rfloor \cdot d \right]_2 \\ &= \left[\sum^S c x_i R^{l_i} \right]_2 - \left[\left\lfloor \sum^S (c x_i R^{l_i} / d) \right\rfloor \right]_2. \end{aligned} \tag{3}$$

The Recrypt process can then be divided into two parts. First we compute the sum of $c x_i R^{l_i}$ for each "block" i. To further optimize this process, encode l_i to a $0-1$ vector $\{\eta_1^{(i)}, \eta_2^{(i)}, \ldots, \eta_n^{(i)}\}$ where only two elements are "1" and all other elements are "0"s. Suppose the two positions are labeled as a and b. We write $l(a, b)$ to refer to the corresponding value of l. Alternatively we can obtain $c x_i R^{l_i}$ from

$$c x_i R^{l_i} = \sum_a \eta_a^{(i)} \sum_b \eta_b^{(i)} c x_i R^{l(a,b)}. \tag{4}$$

Only when $\eta_a^{(i)}$ and $\eta_b^{(i)}$ are both "1", the corresponding $c x_i R^{l(a,b)}$ is selected. In addition, if we encode l in a way that each iteration only increases it by 1, the next factor $c x_i R^{l(a,b)}$ can be easily computed by multiplying R to the result of the previous computation.

After applying these modifications, all operations involved in this formulation of decryption become bit operations realizable by sufficiently shallow circuits. Thus we can evaluate this process homomorphically. The parameters η_i are stored in encrypted form and incorporated into the public key.

2.1 Concrete Parameters

In [3] Gentry and Halevi propose concrete parameter choices for three security settings as shown in Table 1. Note that the parameters, e.g. the determinant d (the modulus used in the FHE primitives) is in the million-bit range and therefore unusually large even for cryptographic applications.

3 Achieving Efficient Large Number Additions

3.1 Addition and Modular Addition

Large number addition has low arithmetic intensity, however, a traditional carry-ripple adder would result a very large

TABLE 1
Parameter Sizes as Proposed by Gentry and Halevi [3]

SETTING	DIMENSION n	DETERMINANT SIZE $\lvert d\rvert$
Small	2048	785006
Medium	8192	3148249
Large	32768	12625500

TABLE 2
Performance of Modular Addition on CPU and GPU

Size in K bits	Modular ADD on CPU	Modular ADD on GPU
1024	0.032 ms	0.016 ms
2048	0.069 ms	0.022 ms
4096	0.139 ms	0.031 ms
8192	0.526 ms	0.048 ms
16384	1.240 ms	0.082 ms

carry-chain that is not efficient for parallel processing. Instead, carry-lookahead addition is often used in for high-speed addition [9]. For parallel processing on a GPU, large number addition is implemented using a carry-lookahead scheme similar to the one presented in [10]. A large number is first broken into m warps, each warp contains n words, and each word is a 32-bit integer. The addition is computed using two GPU kernels. In the first kernel, each warp performs an addition and all warps run in parallel threads. Each warp outputs 3 values: the sum z_i, a carry out bit c_i and a "critical flag" bit f_i. The critical flag is set that if z_i are all 1's, which indicates it will generate a carry out bit if there is a carry in from the adjacent lower warp. The second kernel resolves the whole carry chain for all warps. The second kernel does the hierarchical carry computation to determine the carry in for each chunk. If there is a carry in, then the chunk is incremented, rippling the carry through the chunk as needed.

The modular addition operation, i.e. $x + y \bmod m$, is also implemented as a pair of kernels on a GPU. The first kernel computes $x + y - m$ of the most significant 64 words of x, y and m. If the result is positive, then we only need compute $x + y - m$ of the full value. If the result is strongly negative, then we need only compute $x + y$ of the full value. However, it is possible that the sign of $x + y - m$ cannot be determined from the top 64 words only. In this case, both $x + y$ and $x + y - m$ are computed and stored, while leaving the second kernel to choose one of them. In the first kernel, each warp in the computation handles a chunk of the computation, ignoring the carries from less significant blocks.

The second kernel has all of the inter-block carry logic to resolve the global carries. If the first kernel cannot determine whether $x + y$ or $x + y - m$ to be calculated, the second kernel will firstly resolves the global carries for $x + y - m$. If the result of $x + y - m$ is positive or 0, then the second kernel will give the $x + y - m$ for the final results. If the result of $x + y - m$ is negative, then the second kernel resolves the global carries for $x + y$ and return the result of $x + y$ as the final value. Table 2 shows the computation time of modular additions on CPU and GPU for a few selected operand sizes. Both operands in the test are uniformly sampled and are smaller than the modulus. As mentioned earlier, the CPU platform is Intel Core i7 3770 K processor at 3.5 GHz with 8 GB memory and the GPU is a NVIDIA GTX 690 module plugged into the system.

TABLE 3
Performance of Modular Subtraction on CPU and GPU

Size in K bits	Modular SUB on CPU	Modular SUB on GPU
1024	0.020 ms	0.0163 ms
2048	0.042 ms	0.023 ms
4096	0.111 ms	0.034 ms
8192	0.281 ms	0.054 ms
16384	0.579 ms	0.094 ms

3.2 Subtraction and Modular Subtraction

Our implementation of multi-precision subtraction is very similar to that of addition. Two kernels are used for the CUDA implementation to calculate the $x + \overline{y} + 1$. The operation of the first kernel is similar to the first kernel in modular addition (described above), designed to compute $x_i + \overline{y_i}$. The second kernel is used to resolve the inter-block carries, resulting the final result $x + \overline{y} + 1$.

For modular subtraction on GPU, a pair of kernels are used. The first kernel computes $x - y$ of the most significant 64 words of x, y. If the result is positive, then we only need compute $x - y$ of the full value. If the result is strongly negative, then we need only compute $x - y + m$ of the full value. In the case that the sign of $x - y$ cannot be determined by the top 64 words only, both $x - y$ and $x - y + m$ are computed and it leaves the second kernel to choose one of them. In the first kernel, each warp in the computation handles a chunk of the computation, ignoring the carries from less significant blocks. The second kernel handles all the inter-block carries to resolve the global carries. If the first kernel cannot determine whether $x - y$ or $x - y + m$ is to be calculated, the second kernel will choose the correct value after it resolves the global carries, which is similar to the method used in the modular addition. Table 3 lists the computing time of modular subtraction on CPU and GPU, respectively.

4 MODULAR MULTIPLICATION OF LARGE NUMBERS

Large integer multiplication is by far the most time consuming operation in the FHE primitives which use unusually large parameters summarized in Table 1. Therefore, it becomes the main target for acceleration.

Efficient Montgomery multiplication on GPUs was demonstrated by Bernstein *et al.* in the context of ECM factorization [11]. The authors implement 280-bit modular multiplication on a GPU using the schoolbook method followed by Montgomery multiplication. For short operands with less than a thousand bits this is perfectly suitable. However in our setting, this approach will not yield an efficient implementation. More concretely, assuming a 32-bit word multiplications for 2^{20} bits or equivalently 2^{15} word operands we would need in the order of 2^{30} multiplications (ignoring small constants, memory load/communication cost etc.).

Clearly we need to seek asymptotically more efficient techniques. In [12] Emeliyanenko proposed a technique based on the number theoretical transform for the multiplication of polynomials with integer coefficients on GPUs. The implementation is capable of multiplying large integers by treating their words as polynomial coefficients. The implementation outperforms GMP [24] and NTL [23] for moderate operand

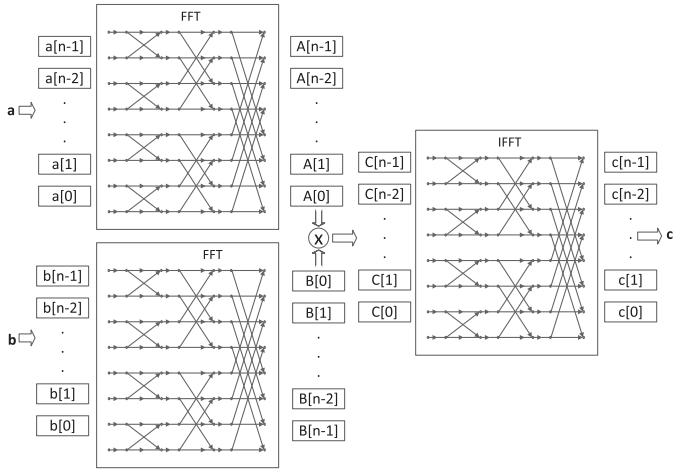

Fig. 1. Strassen's FFT multiplication algorithm.

sizes. Unfortunately, the implementation does not yet support million bit multiplication. The largest operand size supported by the implementation is only 32,768 bits. It takes about 76 ms to complete a multiplication of such length.

In this work we focus on a more recent implementation by Emmart and Weems [14] which is capable of supporting million bit multiplication. We first summarize the Schönhage-Strassen Multiplication Algorithm—the basis of Emmart and Weems' implementation.

4.1 Schönhage-Strassen Multiplication Algorithm

In [13], Strassen described a multiplication algorithm based on Fast Fourier Transform (FFT), which offers a good solution for effectively parallel computation of the large-number multiplication as shown in Fig. 1. The algorithm uses Fast Fourier transforms in rings with $2^{2^n}+1$ elements, i.e. a specialized number theoretic transform. Briefly, the Strassen FFT algorithm can be summarized as follows:

1. Break large numbers A and B into a series of words $a(n)$ and $b(n)$ given a base b, and compute the FFT of the A and B series by treating each word as an sample in the time domain.
2. Multiply the FFT results, component by component: set $C[i] = \mathsf{FFT}(A)[i] * \mathsf{FFT}(B)[i]$.
3. Compute the inverse fast Fourier transform: set $c(n) = \mathsf{IFFT}(C)$.
4. Resolve the carries: when $c[i] \geq b$, set $c[i+1] = c[i+1] + (c[i] \text{ div} b)$, and $c[i] = c[i] \bmod b$.

4.2 Emmart and Weems' Approach

In [14], Emmart and Weems implemented the Strassen FFT based multiplication algorithm on GPUs. Specifically, they performed the FFT operation in finite field Z/pZ with a prime p to make the FFT exact. In fact, they chose the $p = 0\ \text{xFFFFFFFF00000001}$ from a special family of prime numbers which are called Solinas Primes [15]. Solinas Primes support high efficiency modulo computations and this p especially is ideal for 32-bit processors, which has also been incorporated into the latest GPUs. In addition, an improved version of Bailey's FFT technique [16] is employed to compute the large size FFT. The performance of the final implementation is promising. For operands up to 16,320 K bits, it has a speedup of up to 16.7 when comparison with multiplication on the CPUs of the same technology.

TABLE 4
Performance Comparison of Multiplication on CPUs vs. GPUs

Size in K bits	On CPU	On GPU	Speedup
1024 x 1024	8.5 ms	0.583 ms	14.6
2048 x 2048	15.1 ms	1.085 ms	13.9
4096 x 4096	30.4 ms	2.351 ms	12.9
8192 x 8192	63.1 ms	4.850 ms	13.0
16384 x 16384	137.3 ms	8.835 ms	16.7

We follow the implementation in [14]. As seen from Table 4 we gain a significant speedup over the implementations achieved on CPUs. As we can see from Table 4, the actual speedup factors are slightly different from [14]. Nevertheless, it is a significant speedup over the implementations achieved on CPUs. Therefore, we employ this specific instance of the Strassen FFT based multiplication algorithm in the FHE implementation.

We note, however, the optimizations we later introduce virtually eliminate the need for FFT conversions except at the very beginning or the very end of the computation chains. The only exception is the decryption primitive which is implemented using a single modular multiplication. Therefore, while still necessary, the efficiency of the FFT operation has a negligible impact on the overall performance of encryption and recryption.

4.3 Modular Multiplication

Efficient modular multiplication is crucial for the decryption primitive. The other primitives only use modular reduction at the very end of the computations only once. Many cryptographic software implementations employ the Montgomery multiplication algorithm, cf. [17], [18]. Montgomery multiplication replaces costly divisions with additional multiplications. Unfortunately, the interleaved versions of the Montgomery multiplication algorithm generates long carry chains with little instruction-level parallelism. For the same reason, it is hard to implement the interleaved Montgomery multiplication algorithm on parallel computing friendly GPUs. For example, a Montgomery multiplication implementation on a GeForce 9800GX2 card was presented in [19]. The speedup factor of GPU decreased from 2.6 to 0.6 when the operand size increases from 160-bit to 384-bit, which showed little speedup if any can be achieved with large operand sizes. In addition, the underlying large integer multiplication algorithm we use is FFT based and optimized for very large numbers. Therefore, there does not seem to be any easy way to break it into smaller pieces. In conclusion, we implement modular multiplications without integrating the multiplication and reduction steps, but instead by executing them in sequence. With this approach long carry chains without significant instruction-level parallelism can be broken into multiple independent carry chains. This increases the overall number of carries, but may be faster on parallel hardware.

4.3.1 Modular Reduction

The most popular algorithms for modular reduction are the Montgomery reduction [20] and the Barrett reduction

algorithms [21]. As mentioned earlier, the interleaved Montgomery reduction algorithm cannot exploit the parallel processing on GPUs. The Barrett approach has a simpler structure and thus lends itself better for further optimizations. Therefore, we select the Barrett method to implement modular reductions.

Given two positive integers t and M, the Barrett modular reduction approach computes $r = t \bmod M$. A version of Barrett's reduction algorithm is shown in Algorithm 1. It can be shown that the initial r for the loop in lines 5-7 is smaller than $3M - 1$. Therefore, the loop can finish quickly. In addition, the value $\mu = \lfloor \frac{2^q}{M} \rfloor (q = 2\lceil \log_2(M) \rceil)$ can be precomputed to speed up the process. If multiple reductions are to be computed with the same modulus M. Then this value can be reused for all reductions, which is exactly the case we have.

Algorithm 1 Barret Reduction Algorithm

1: **procedure** BARRETT(t, M) ▷ Output: $r = t \bmod M$
2: $q \leftarrow 2\lceil \log_2(M) \rceil$ ▷ Precomputation
3: $\mu \leftarrow \lfloor \frac{2^q}{M} \rfloor$ ▷ Precomputation
4: $r \leftarrow t - M\lfloor t\mu / 2^q \rfloor$
5: **while** $r \geq M$ **do**
6: $r \leftarrow r - M$
7: **end while**
8: **return** r ▷ $r = t \bmod M$
9: **end procedure**

In addition, it would be advantageous to apply truncations only at multiples of the word size w of the multiplier hardware (usually 32 bits) rather than at the original bit positions. In this case, we require q to be a multiple of the word size w. With this approach, the division by 2^q can be easily implemented by discarding the least significant q/w words.

5 Optimization of FHE Primitives

The FHE algorithm consists of four primitives: KeyGen, Encrypt, Decrypt and Recrypt. The KeyGen is only called once during the setup phase. Since keys are generated once and then preloaded to the GPU, the speed of KeyGen is not as important. Therefore we focus our attention to optimizing the other three primitives.

For the Decrypt primitive, we perform the computation as in Eq. 2. The flow is shown in Fig. 2. Obviously, the time spent in the primitive is equivalent to the time it takes to compute a single modular multiplication with large operands. Applying the FFT based Strassen algorithm and Barrett reduction, which we discussed earlier, yields significant speedup for the **Decrypt** operation.

5.1 Optimizing Encrypt

To implement the Encrypt primitive, we need to evaluate a degree-$(n-1)$ polynomial u at point r. In [3], a recursive approach for evaluating the binary polynomial u, i.e. $u_i \in \{0, 1\}$, of degree $(n-1)$ at root r modulo d. The

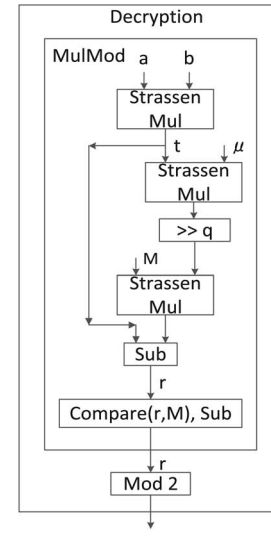

Fig. 2. Decryption procedure.

summation $u(r) = \sum_{i=0}^{n-1} u_i r^i$ is split into a "bottom half" $u^{bot}(r) = \sum_{i=0}^{n/2-1} u_i r^i$ and a "top half" $u^{top}(r) = \sum_{i=0}^{n/2-1} u_{i+n/2} r^i$. Then $y = r^{n/2} u^{top}(r) + u^{bot}(r)$ can be computed. The same procedure repeats until the remaining degree is small enough to be computed directly.

In our implementation, to fully exploit the power of precomputation, we use a direct approach for polynomial evaluations. Specifically, we apply the sliding window technique to compute the polynomial. Suppose the window size is w and we need $t = n/w$ windows, we compute:

$$\sum (u_i r^i) = \sum_{j=0}^{t-1} \left[r^{w \cdot j} \cdot \sum_{i=0}^{w-1} (u_{i+wj} r^i) \right]. \quad (5)$$

Here all additions and multiplications are evaluated modulo d. After organizing the computation as described above, we can introduce precomputation to speed up the process. As r is determined during KeyGen and therefore known apriori, the $r^i, i = 0, 1, \ldots, w$ values can be precomputed. In order to further reduce the overhead caused by the relatively slow communication between the CPU and the GPU, these precomputed values can be preloaded into GPU memory before the Encrypt process starts. A larger window size w leads to fewer multiplications but an increased memory requirement. Hence, we have a trade-off between speed and memory use. In addition, as mentioned in previous sections, the majority of the coefficients of u are zeros. It is possible that all the coefficients a window are zeros. In this case, we can skip the multiplication to further speedup.

We summarized this approach in a short paper [22]. We would like to stress that the technique outlined in [22] does not fully take advantage of the FFT domain representations since FFT based arithmetic is restricted to within the boundaries of modular multiplication. More explicitly in each modular multiplication first the operands are taken through the FFT transformation, then this is followed by component-wise multiplications and the inverse FFT computation and modular reduction. The evaluation of one window during encryption is shown in Fig. 3. For ease of reference, we refer to this approach as the **partial FFT optimization**.

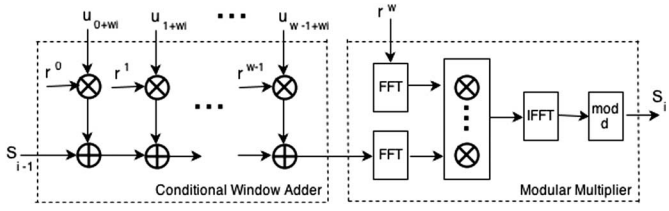

Fig. 3. Partial-FFT domain evaluation of one window of Encrypt.

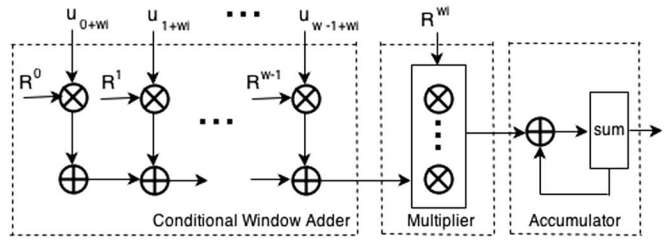

Fig. 4. Full-FFT domain window evaluation and accumulation in Encrypt.

Here we take a step further. Since we are using an FFT based algorithm to compute the multiplications, the precomputed values can also be saved in FFT form. Even further, since the FFT form is linear, we can directly evaluate additions in FFT domain. Therefore, the whole computation before the final reduction can be performed in FFT domain:

$$\sum(u_i r^i) = \mathsf{IFFT}\left(\sum_{j=0}^{n/w-1}\left[R^{w\cdot j}\cdot\sum_{i=0}^{w-1}(u_{i+wj}R^i)\right]\right)(\mathrm{mod}\ d), \quad (6)$$

where R^i is the precomputed FFT form of corresponding r^i. With this reformulation we eliminated almost all of the costly FFTs and IFFTs and modular reductions (compare Figs. 3 and 4). Also as a side-benefit of staying in the FFT domain, carry propagations among words normally performed during addition operations are eliminated, which also contributes to the speed up.

The side-effect of staying in the FFT domain is that we need to ensure that there is enough space in the FFT representation to prevent overflow. Also intermediary computation results need to be kept in double size buffers. To prevent excessive growth in the FFT coefficients we can no longer keep multiplying the intermediary result by r^w. Instead, we need to keep the precomputed R^{wi} for $i = 0\ldots,(n/w-1)$ values in storage. With this simple trick we can now implement Encrypt by evaluating a shallow (depth 2) circuit. Note that computing differences of variables in the FFT domain may yield incorrect results unless we compute subtractions as $\mathsf{FFT(x-y)} = \mathsf{FFT(x)} + \mathsf{FFT(d-y)}$. For this, we need to precompute both r^i and $d - r^i$ for $i = 0,\ldots,w-1$. The final algorithm is depicted in Fig. 4.

5.2 Implementing Recrypt

The Recrypt primitive is significantly more complicated than Encrypt. As mentioned earlier, Recrypt can be divided into two steps: processing of S blocks and the computation of their sum. In the first step, the most time-consuming computation is as follows

$$cx_i R^{l_i} = \sum_a \eta_a^{(i)} \sum_b \eta_b^{(i)} cx_i R^{l(a,b)}. \quad (7)$$

Here η_i is part of the public key. If we encode the l in a proper way such that each iteration it only increases by one, the next factor $cx_i R^{l(a,b)}$ can be easily computed by multiplying R with the result of the previous iteration. Here we refer to $cx_i R^{l(a,b)}$ as the $factor$. In each iteration, we update $factor \cdot R \bmod d$ and determine whether we should add η_b or not. Since R is a small constant, the computation may even be performed on the CPU without any noticeable loss of efficiency. Therefore, the CPU is used to compute the new $factor$ value while the GPU is busy computing the additions from previous iteration. This approach allows us to run the CPU and the GPU concurrently and therefore harnessing the full computational power of the overall system.

The constants used in Recrypt are part of the public key. They can be precomputed to further speed up the computation. Similar to Encrypt, the public keys can be pre-loaded into the GPU memory to eliminate the latency incurred in CPU-GPU communications. For the small parameter setting the public key is about 140 MB. The public key can perfectly fit into the GPU memory of the latest graphic cards. In fact, the public key will fit into the GPU memory even in the large setting, whose public key is about 2.25 GB [3]. This initial implementation [22] of Recrypt is shown in Fig. 5.

Again we take a step further. The majority of the computation in the $Recrypt$ can be represented as an "add-mul-add" chain. Therefore, similar optimizations can be applied. The computation before reduction can be performed in the FFT domain, reducing the number of expensive FFT and IFFT operations significantly. However, this optimization will also cause growth in public key and make it impossible to store the whole public key in the GPU memory in the medium dimension case. Fortunately, for the medium settings, the computation time is long enough to dwarf this extra communication overhead. The optimized implementation of Recrypt is depicted in Fig. 6.

6 Implementation Results

We implemented the Encrypt, Decrypt and Recrypt primitives of the Gentry-Halevi FHE scheme with the proposed optimizations on two GPU machines[1]:

- **Tesla C2050**: Intel Xeon X5650 2.67 GHz, 14 GB RAM and two NVIDIA Tesla C2050s, each has 448 cores, with 3 GB memory running at 1150 MHz,
- **GTX 690**: Intel Core i7 3770 K running at 3.5 GHz with 8 GB RAM and a NVIDIA GTX 690 housing two GPUs each with 768 cores running at 1.02 GHz with 4 GB memory.

Of the available two GPU processors only one is used on both platforms. Shoup's NTL library [23] is used for high-level numeric operations and GNU's GMP library [24] for the underlying integer arithmetic operations. A modified version of the code from [14] is used to perform the Strassen FFT multiplication on the GPUs.

1. The GPU software is made available for download at http://vernam.wpi.edu.

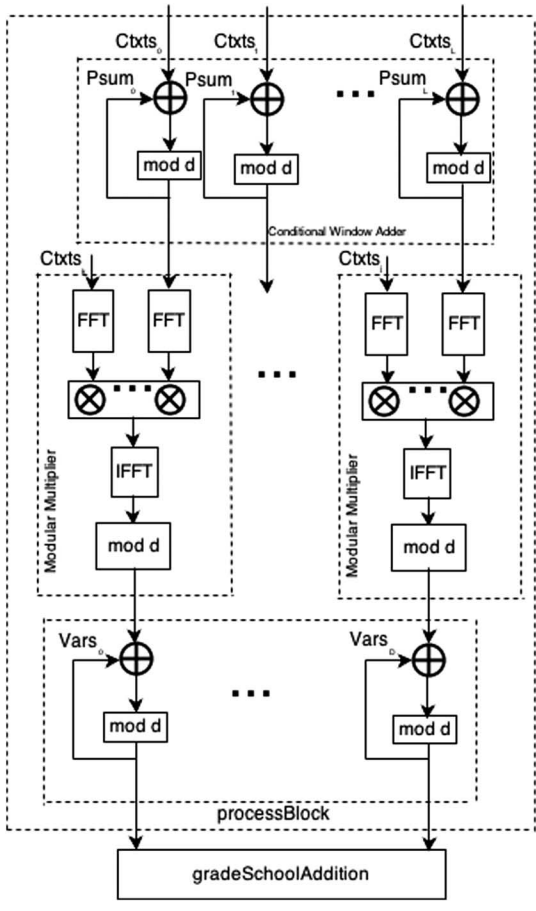

Fig. 5. Partial-FFT domain evaluation of Recrypt.

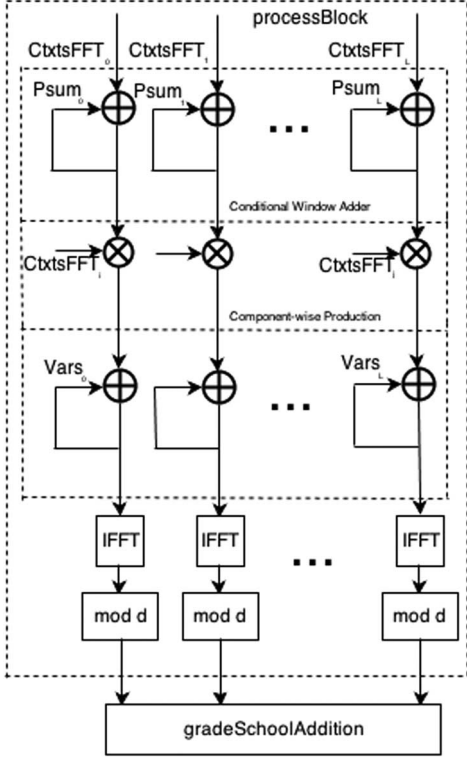

Fig. 6. Full-FFT domain evaluation of Recrypt.

TABLE 5
Performance Comparison of FHE Primitives with Partial FFT [22] vs. Full FFT Optimizations on Tesla C2050

Small Setting: Dimension 2048			
Operation	Partial-FFT	Full-FFT	Speedup
Encrypt	220 msec	4.9 msec	44
Decrypt	2.5 msec	2.5 msec	1
Recrypt	4.2 sec	1.83 sec	2.3
Medium Setting: Dimension 8192			
Operation	Partial-FFT	Full-FFT	Speedup
Encrypt	1.03 sec	37 msec	27
Decrypt	12.9 msec	12.9 msec	1
Recrypt	21.3 sec	12.4 msec	1.7

6.1 Partial versus Full FFT Domain Evaluation

To compare the performance of the two optimizations we introduced, i.e. the partial FFT [22] and the full-FFT optimization we implemented both schemes with small and medium parameter settings with dimensions of 2,048 and 8192, respectively, on the C2050 platform. In our implementation we choose the window size as $w = 64$. The performance results are summarized in Table 5.

As expected, the decryption times are identical since decryption involves only a single modular multiplication. On the other hand, by delaying modular reductions and thereby removing the intermediary FFT/IFFT transformations, and by introducing aggressive FFT domain precomputation we significantly improved the performance of the encryption algorithm. More significantly, the performance for Recrypt also improves significantly by 2.3 times for the small case. The speedup is diminished for the medium dimension to 1.7 times. This can be explained by the fact that the FFT optimization in Recrypt does not extend to the grade school addition part of the computation. Since recryption speed is the most critical problem in FHE, the comparison above clearly shows that the Full FFT approach is superior. Therefore, we focus on this optimization alone in the remainder of this section.

This improvement comes at the expense of increased memory use to keep the precomputed FFT domain powers of R. The storage required to keep precomputated values is summarized in Table 6. For the partial FFT optimization, to support Encrypt we need to maintain $w + 1$ powers of r. In the implementation we used $w = 64$. For Recrypt we need to preload the public keys into the GPU memory. In the full FFT case, we need to keep these values, i.e. powers of r and the public keys, in FFT form. Since we double the length of the operands and each 16-bit input coefficient is stored in a 64-bit coefficient each operand requires 8 times as much memory. For Encrypt, besides the r^i powers and their complements, we also need to keep the n/w powers of r^{iw} where $i = 0, \ldots, n/w - 1$ in FFT form. This means we need to keep $2(n/w + w)$ operands in FFT form for full FFT Encrypt.

6.2 Full-FFT Domain Results

Here we focus only on implementations of the full-FFT optimization on both GPU platforms, and compare their performance to that of the original implementation provided by Gentry and Halevi [3]. For fairness, we recompiled the code provided by Gentry and Halevi for the CPU implementation [3] on the GTX 690 Intel i7 CPU for comparison.

As clearly seen in Table 7, our implementation for the small case is about 174, 7.6 and 13.5 times faster than the original

TABLE 6
Memory Use with Partial FFT [22] vs. Full FFT Optimizations on Tesla C2050

Small Setting: Dimension 2048		
Operation	Partial-FFT	Full-FFT
Encrypt	8 Mbytes	192 Mbytes
Recrypt	86 Mbytes	688 Mbytes
Medium Setting: Dimension 8192		
Operation	Partial-FFT	Full-FFT
Encrypt	32 Mbytes	1.5 Gbytes
Recrypt	352 Mbytes	2.8 Gbytes
Large Setting: Dimension 32768		
Operation	Partial-FFT	Full-FFT
Encrypt	129 Mbytes	18 Gbytes
Recrypt	2.5 Gbytes	20 Gbytes

TABLE 7
Performance of FHE Primitives with Full-FFT Optimization vs. CPU Implementation [3]

Small Setting: Dimension 2048				
Operation	CPU	C2050	GTX 690	Speedup
Encrypt	1.08 sec	6.3 msec	6.2 msec	174
Decrypt	14 msec	2.5 msec	1.84 msec	7.6
Recrypt	17.8 sec	1.8 sec	1.32 sec	13.5
Medium Setting: Dimension 8192				
Operation	CPU	C2050	GTX 690	Speedup
Encrypt	10.6 sec	37 msec	24 msec	442
Decrypt	70 msec	13 msec	7.2 msec	9.7
Recrypt	96.3 sec	12.4 msec	8.4 sec	11.5
Large Setting: Dimension 32768				
Operation	CPU	C2050	GTX 690	Speedup
Encrypt	82 sec	13.5 sec	4.1 sec	20
Decrypt	295 msec	90 msec	29 msec	10.1
Recrypt	800 sec	TBD	235 sec	3.4

Gentry-Halevi code for encryption, decryption and recryption, respectively [3] on the GTX 690 platform. The impressive speedup of encryption is due to the fact that encryption benefits significantly from precomputation. For the medium case with dimension 8192, we also achieved a speed up of 442, 9.7 and 11.7. Note that the encryption process enjoys even more speedup as the dimension grows.

To explore the effect of our optimizations, we also broke down and evaluated the time consumption for the Recrypt primitive. In the small case, for instance, if we look into the 1.32 seconds of time it takes to compute the Recrypt, we discover that it takes about 0.87 seconds for processing blocks and 0.46 seconds for grade-school addition. Further inspection of the block processing part reveals that the GPU multiplications and additions take about 0.54 second. In the meantime, it takes the CPU about 0.6 second to compute the factor. Clearly, the sum of this two latencies amounts to more than 0.87 seconds. This is due to the fact that the CPU and the GPU are working in parallel. In addition, the time consumed on multiplication is minimized due to the drastic reduction in the number of costly FFT/IFFT computations and modular reductions.

6.3 Memory Issues in the Large Dimensions

We also evaluated the prospects of supporting the large dimension setting. The arithmetic operations in our implementation are based on the FFT algorithm and their complexity grows with $O(n\log(n))$ with the dimension n. In fact, since we remove most of the IFFT/FFTs the complexity is almost linear, i.e. $O(n)$. Thus, the algorithm itself is scalable and can support larger parameter sets. However, in the implementation the GPU memory becomes the bottleneck that restricts the dimensions with which we can compute the FHE primitives efficiently.

The encryption scheme relies heavily on precomputation. If all the pre-computed values can be loaded into the GPU memory, the algorithm can run very efficiently. However, as the operands and the dimension grows, the size and number of the pre-computed values also grows. When the GPU memory cannot hold all pre-computed values, we have to keep parts of them in the main memory and load them when necessary. The GPU memory will have to update frequently and the communication overhead will slow down the algorithm significantly. The recryption process shares the same problem. The public keys need to be loaded into GPU memory to support efficient recryption process. When the public keys did no longer fit into the GPU memory, we had to use the GPU memory essentially in the role of a slow cache.

We need a large GPU memory to support efficient implementation with larger parameters. With 3 GB and 2 GB (per core) memories for our Tesla and GTX 690 GPU platforms, respectively, can only support the small and medium parameter sizes efficiently (see Table 6). For instance, in the medium case, the GPU memory is not large enough to hold the whole public key. Therefore, the keys are loaded when required, which introduces an overhead of about 0.6 seconds for Recrypt on the GTX 690. However, as the computation for the medium case consumes much more time, the impact of this overhead to the overall performance is limited.

For the large setting the situation gets worse since the memory has to be reloaded several times during the computation of the Encrypt and Recrypt primitives partially loosing the advantage of precomputation and preloading of public keys.

7 Conclusion

In this paper we tackled the performance bottleneck of the Gentry Halevi FHE by introducing an array of algorithmic optimizations. Specifically, we presented two optimizations: partial FFT where we speed up modular multiplications by introducing Strassen's FFT based multiplication algorithm, and full FFT optimization where we gain additional speed by eliminating the majority of FFT conversions via delayed modular reductions and by performing the bulk of the computations directly in the FFT domain. In both optimizations we employed an aggressive precomputation strategy.

We implemented both optimizations on NVIDIA GTX 690 and the Tesla C2050 GPU platforms for all three parameter settings. Experimental results with small parameter setting (2048 dimension) yield two and one order of magnitude speed for Encrypt and Recrypt, respectively. We observed further improvement in the speedup in the medium parameter setting. Not suprisingly, in the large setting our precomputation/preloading strategy breaks down and memory becomes the bottleneck loosing some of the speed gains.

Acknowledgments

This work was supported in part by the National Science Foundation under CNS Award #1117590.

References

[1] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," *Proc. 41st Ann. ACM Symp. Theory of Computing*, pp. 169-178, 2009.

[2] R. Rivest, L. Adleman, and M. Dertouzos, "On Data Banks and Privacy Homomorphisms," *Foundations of Secure Computation*, vol. 32, no. 4, pp. 169-178, 1978.

[3] C. Gentry and S. Halevi, "Implementing Gentry's Fully-Homomorphic Encryption Scheme," *Proc. Advances in Cryptology-EUROCRYPT 2011*, pp. 129-148, 2011.

[4] D.B. Cousins, K. Rohloff, C. Peikert, and R. Schantz, "SIPHER: Scalable Implementation of Primitives for Homomorphic Encryption—FPGA Implementation Using Simulink," *Proc. High Performance Extreme Computing Conf.*, 2011.

[5] X. Cui, Y. Chen, and H. Mei, "Improving Performance of Matrix Multiplication and FFT on GPU," *Proc. 15th Int'l Conf. Parallel and Distributed Systems (ICPADS)*, pp. 42-48, 2009.

[6] N. Govindaraju, N. Raghuvanshi, and D. Manocha, "Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 611-622, 2005.

[7] J.B. Pezoa, D. Fasoli, and O. Faugeras, "Three Applications of GPU Computing in Neuroscience," *Computing in Science and Eng.*, vol. 14, no. 3, pp. 40-47, 2011.

[8] C. Gentry, "A Fully Homomorphic Encryption Scheme," PhD dissertation, Stanford Univ., 2009.

[9] Y. Pai and Y. Chen, "The Fastest Carry Lookahead Adder," *Proc. Second IEEE Int'l Workshop on Electronic Design, Test and Applications (DELTA'04)*, pp. 434-436, 2004.

[10] N. Emmart and C. Weems, "High Precision Integer Addition, Subtraction and Multiplication with a Graphics Processing Unit," *Parallel Processing Letters*, vol. 20, no. 4, pp. 293-306, 2010.

[11] D.J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang, "ECM on Graphics Cards," in *EUROCRYPT 2009*, vol. 5479, A. Joux, Ed. New York, NY, USA: Springer, 2010, pp. 483-501.

[12] P. Emeliyanenko, "Efficient Multiplication of Polynomials on Graphics Hardware," *Proc. Advanced Parallel Processing Technologies*, pp. 134-149, 2009.

[13] A. Schönhage and V. Strassen, "Schnelle Multiplikation Großer Zahlen," *Computing*, vol. 7, no. 3, pp. 281-292, 1971.

[14] N. Emmart and C. Weems, "High Precision Integer Multiplication with a GPU Using Strassen's Algorithm with Multiple FFT Sizes," *Parallel Processing Letters*, vol. 21, no. 3, p. 359, 2011.

[15] J. Solinas, "Generalized Mersenne Numbers," Tech. rep., 1999, http://www.cacr.math.uwaterloo.ca/techreports/1999/corr99-39.ps

[16] D. Bailey, "FFTs in External of Hierarchical Memory," *Proc. ACM/IEEE Conf. Supercomputing*, pp. 234-242, 1989.

[17] C. Mclvor, M. McLoone, and J. McCanny, "Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures," *Proc. Record 37th Asilomar Conf. Signals, Systems and Computers*, vol. 1, pp. 379-384, 2003.

[18] A. Daly and W. Marnane, "Efficient Architectures for Implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic," *Proc. ACM/SIGDA 10th Int'l Symp. Field-Programmable Gate Arrays*, pp. 40-49, 2002.

[19] P. Giorgi, T. Izard, and A. Tisserand, "Comparison of Modular Arithmetic Algorithms on GPUs," in Proc. Int. Conf. Parallel Comput. (ParCo'09), Lyon, France, Sep. 2009.

[20] P. Montgomery, "Modular Multiplication without Trial Division," *Math. Computation*, vol. 44, no. 170, pp. 519-521, 1985.

[21] P. Barrett, "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor," *Proc. Advances in Cryptology (CRYPTO'86)*, 1987, pp. 311-323.

[22] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating Fully Homomorphic Encryption Using GPU," *Proc. IEEE High Performance Extreme Computing Conf. (HPEC'12)*, pp. 1-5, 2012.

[23] V. Shoup, "NTL: A Library for Doing Number Theory," Version 5.5.2, 2010, http://shoup.net/ntl/

[24] *The GNU Multiple Precision Arithmetic Library, Version 5.0.1*, http://gmplib.org/, 2010.

Wei Wang received the BE degree from Shandong University, Jinan, China, in 2007, and the ME degree from Tsinghua University, Beijing, China, in 2010. He is currently working toward the PhD degree in the Electrical and Computer Engineering Department at Worcester Polytechnic Institute, Massachusetts. His research interests focus on circuit and system designs for fully homomorphic encryption and RSA cryptosystems.

Yin Hu received the bachelor's degree with a double major in electrical engineering and applied physics from Shanghai Jiaotong University, China, in 2007, and the MS degree in electrical and computer engineering (ECE) from Worcester Polytechnic Institute, Massachusetts. He received the PhD degree on homomorphic encryption in May 2013. His research interest includes security in cloud computing, memory authentication, and homomorphic encryption.

Lianmu Chen received the bachelor's degree with a major in communications engineering from Beijing University of Posts and Telecommunications, Beijing, China, in 2011, and the MS degree in electrical and computer engineering from Worcester Polytechnic Institute, Massachusetts, in 2013. He is currently working at Unified Storage Division, EMC. His research is focused on cryptography and homomorphic encryption.

Xinming Huang (M'01-SM'09) received the PhD degree in electrical engineering from Virginia Tech, Blacksburg, in 2001. He is currently an associate professor at Worcester Polytechnic Institute (WPI), Massachusetts. From 2001 to 2003, he was a member of technical staff with the wireless advanced technology laboratory, Bell Labs of Lucent Technologies, New Jersey. His research interests are in the areas of circuits and systems, with emphasis on reconfigurable computing, wireless communications, and embedded systems.

Berk Sunar received the BSc degree in electrical and electronics engineering from Middle East Technical University, Turkey, in 1995, and the PhD degree in electrical and computer engineering from Oregon State University, Corvallis, in 1998. In Fall 2000, he joined Worcester Polytechnic Institute, Massachusetts, where since July 2006 he has served as an associate professor. He is heading the WPI Vernam Security Laboratory. He received numerous awards including the National Science Foundation CAREER award in 2002, and the IBM Research Pat Goldberg Best Paper award in 2007 for his work on Trojan Detection. He is a member of the International Association of Cryptologic Research (IACR) professional societies. His research interests are in the applications of cryptography.