Homomorphic Evaluation of the AES Circuit

Craig Gentry[1], Shai Halevi[1], and Nigel P. Smart[2]

[1] IBM Research
[2] University of Bristol

Abstract. We describe a working implementation of leveled homomorphic encryption (without bootstrapping) that can evaluate the AES-128 circuit in three different ways. One variant takes under over 36 hours to evaluate an entire AES encryption operation, using NTL (over GMP) as our underlying software platform, and running on a large-memory machine. Using SIMD techniques, we can process over 54 blocks in each evaluation, yielding an amortized rate of just under 40 minutes per block. Another implementation takes just over two and a half days to evaluate the AES operation, but can process 720 blocks in each evaluation, yielding an amortized rate of just over five minutes per block. We also detail a third implementation, which theoretically could yield even better amortized complexity, but in practice turns out to be less competitive.

For our implementations we develop both AES-specific optimizations as well as several "generic" tools for FHE evaluation. These last tools include (among others) a different variant of the Brakerski-Vaikuntanathan key-switching technique that does not require reducing the norm of the ciphertext vector, and a method of implementing the Brakerski-Gentry-Vaikuntanathan modulus-switching transformation on ciphertexts in CRT representation.

1 Introduction

In his breakthrough result [9], Gentry demonstrated that fully-homomorphic encryption was theoretically possible, assuming the hardness of some problems in integer lattices. Since then, many different improvements have been made, for example authors have proposed new variants, improved efficiency, suggested other hardness assumptions, etc. Some of these works were accompanied by implementation [19, 10, 6, 20, 14, 7], but all the implementations so far were either "proofs of concept" that can compute only one basic operation at a time (at great cost), or special-purpose implementations limited to evaluating very simple functions. In this work we report on the first implementation powerful enough to support an "interesting real world circuit". Specifically, we implemented a variant of the leveled FHE-without-bootstrapping scheme of Brakerski, Gentry, and Vaikuntanathan [3] (BGV), with support for deep enough circuits so that we can evaluate an entire AES-128 encryption operation.

Why AES? We chose to shoot for an evaluation of AES since it seems like a natural benchmark: AES is widely deployed and used extensively in security-aware applications (so it is "practically relevant" to implement it), and the AES circuit is nontrivial on one hand, but on the other hand not astronomical. Moreover the AES circuit has a

regular (and quite "algebraic") structure , which is amenable to parallelism and optimizations. Indeed, for these same reasons AES is often used as a benchmark for implementations of protocols for secure multi-party computation (MPC), for example [17, 8, 12, 13]. Using the same yardstick to measure FHE and MPC protocols is quite natural, since these techniques target similar application domains and in some cases both techniques can be used to solve the same problem.

Beyond being a natural benchmark, homomorphic evaluation of AES decryption also has interesting applications: When data is encrypted under AES and we want to compute on that data, then homomorphic AES decryption would transform this AES-encrypted data into an FHE-encrypted data, and then we could perform whatever computation we wanted. (Such applications were alluded to in [14, 20, 4]).

Why BGV? Our implementation is based on the (ring-LWE-based) BGV cryptosystem [3], which at present is one of three variants that seem the most likely to yield "somewhat practical" homomorphic encryption. The other two are the NTRU-like cryptosystem of Lòpez-Alt et al. [15] and the ring-LWE-based fixed-modulus cryptosystem of Brakerski [2]. (These two variants were not yet available when we started our implementation effort.) These three different variants offer somewhat different implementation tradeoffs, but they all have similar performance characteristics. At present we do not know which of them will end up being faster in practice, but the differences are unlikely to be very significant. Moreover, we note that most of our optimizations for BGV are useful also for the other two variants.

Our Contributions. Our implementation is based on a variant of the BGV scheme [3, 5, 4] (based on ring-LWE [16]), using the techniques of Smart and Vercauteren (SV) [20] and Gentry, Halevi and Smart (GHS) [11], and we introduce many new optimizations. Some of our optimizations are specific to AES, these are described in Section 4. Most of our optimization, however, are more general-purpose and can be used for homomorphic evaluation of other circuits, these are described in Section 3.

Many of our general-purpose optimizations are aimed at reducing the number of FFTs and CRTs that we need to perform, by reducing the number of times that we need to convert polynomials between coefficient and evaluation representations. Since the cryptosystem is defined over a polynomial ring, many of the operations involve various manipulation of integer polynomials, such as modular multiplications and additions and Frobenius maps. Most of these operations can be performed more efficiently in evaluation representation, when a polynomial is represented by the vector of values that it assumes in all the roots of the ring polynomial (for example polynomial multiplication is just point-wise multiplication of the evaluation values). On the other hand some operations in BGV-type cryptosystems (such as key switching and modulus switching) seem to require coefficient representation, where a polynomial is represented by listing all its coefficients.[3] Hence a "naive implementation" of FHE would need to convert the polynomials back and forth between the two representations, and these conversions turn out to be the most time-consuming part of the execution. In our implementation we keep ciphertexts in evaluation representation at all times, converting to coefficient representation only when needed for some operation, and then converting back.

[3] The need for coefficient representation ultimately stems from the fact that the noise in the ciphertexts is small in coefficient representation but not in evaluation representation.

We describe variants of key switching and modulus switching that can be implemented while keeping almost all the polynomials in evaluation representation. Our key-switching variant has another advantage, in that it significantly reduces the size of the key-switching matrices in the public key. This is particularly important since the main limiting factor for evaluating deep circuits turns out to be the ability to keep the key-switching matrices in memory. Other optimizations that we present are meant to reduce the number of modulus switching and key switching operations that we need to do. This is done by tweaking some operations (such as multiplication by constant) to get a slower noise increase, by "batching" some operations before applying key switching, and by attaching to each ciphertext an estimate of the "noisiness" of this ciphertext, in order to support better noise bookkeeping.

Our Implementation. Our implementation was based on the NTL C++ library running over GMP, we utilized a machine which consisted of a processing unit of Intel Xeon CPUs running at 2.0 GHz with 18MB cache, and most importantly with 256GB of RAM.[4]

Memory was our main limiting factor in the implementation. With this machine it took us just under two days to compute a single block AES encryption using an implementation choice which minimizes the amount of memory required; this is roughly two orders of magnitude faster than what could be done with the Gentry-Halevi implementation [10]. The computation was performed on ciphertexts that could hold 864 plaintext slots each; where each slot holds an element of $\mathbb{F}_{2^8}$. This means that we can compute $\lfloor 864/16 \rfloor = 54$ AES operations in parallel, which gives an amortize time per block of roughly forty minutes. A second (byte-sliced) implementation, requiring more memory, completed an AES operation in around five days; where ciphertexts could hold 720 different $\mathbb{F}_{2^8}$ slots (hence we can evaluate 720 blocks in parallel). This results in an amortized time per block of roughly five minutes.

We note that there are a multitude of optimizations that one can perform on our basic implementation. Most importantly, we believe that by using the "bootstrapping as optimization" technique from BGV [3] we can speedup the AES performance by an additional order of magnitude. Also, there are great gains to be had by making better use of parallelism: Unfortunately, the NTL library (which serves as our underlying software platform) is not thread safe, which severely limits our ability to utilize the multi-core functionality of modern processors (our test machine has 24 cores). We expect that by utilizing many threads we can speed up some of our (higher memory) AES variants by as much as a 16x factor; just by letting each thread compute a different S-box lookup.

Organization. In Section 2 we review the main features of BGV-type cryptosystems [4, 3], and briefly survey the techniques for homomorphic computation on packed ciphertexts from SV and GHS [20, 11]. Then in Section 3 we describe our "general-purpose" optimizations on a high level, with additional details provided in the full version of the paper. A brief overview of AES and a high-level description and performance numbers is provided in Section 4.

[4] This machine was BlueCrystal Phase 2; and the authors would like to thank the University of Bristol's Advanced Computing Research Centre (`https://www.acrc.bris.ac.uk/`) for access to this facility.

2 Background

For an integer q we identify the ring $\mathbb{Z}/q\mathbb{Z}$ with the interval $(-q/2, q/2] \cap \mathbb{Z}$, and use $[z]_q$ to denote the reduction of the integer z modulo q into that interval. Our implementation utilizes polynomial rings defined by cyclotomic polynomials, $\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$. The ring $\mathbb{A}$ is the ring of integers of a the mth cyclotomic number field $\mathbb{Q}(\zeta_m)$. We let $\mathbb{A}_q \stackrel{\text{def}}{=} \mathbb{A}/q\mathbb{A} = \mathbb{Z}[X]/(\Phi_m(X), q)$ for the (possibly composite) integer q, and we identify $\mathbb{A}_q$ with the set of integer polynomials of degree upto $\phi(m) - 1$ reduced modulo q.

Coefficient vs. Evaluation Representation. Let m, q be two integers such that $\mathbb{Z}/q\mathbb{Z}$ contains a primitive m-th root of unity, and denote one such primitive m-th root of unity by $\zeta \in \mathbb{Z}/q\mathbb{Z}$. Recall that the m'th cyclotomic polynomial splits into linear terms modulo q, $\Phi_m(X) = \prod_{i \in (\mathbb{Z}/m\mathbb{Z})^*} (X - \zeta^i) \pmod q$.

We consider two ways of representing an element $a \in \mathbb{A}_q$: Viewing a as a degree-$(\phi(m) - 1)$ polynomial, $a(X) = \sum_{i<\phi(m)} a_i X^i$, the *coefficient representation* of a just lists all the coefficients in order $\mathbf{a} = \langle a_0, a_1, \ldots, a_{\phi(m)-1} \rangle \in (\mathbb{Z}/q\mathbb{Z})^{\phi(m)}$. For the other representation we consider the values that the polynomial $a(X)$ assumes on all primitive m-th roots of unity modulo q, $b_i = a(\zeta^i) \bmod q$ for $i \in (\mathbb{Z}/m\mathbb{Z})^*$. The b_i's in order also yield a vector $\mathbf{b} \in (\mathbb{Z}/q\mathbb{Z})^{\phi(m)}$, which we call the *evaluation representation* of a. Clearly these two representations are related via $\mathbf{b} = V_m \cdot \mathbf{a}$, where V_m is the Vandermonde matrix over the primitive m-th roots of unity modulo q. We remark that for all i we have the equality $(a \bmod (X - \zeta^i)) = a(\zeta^i) = b_i$, hence the evaluation representation of a is just a polynomial Chinese-Remaindering representation.

In both representations, an element $a \in \mathbb{A}_q$ is represented by a $\phi(m)$-vector of integers in $\mathbb{Z}/q\mathbb{Z}$. If q is a composite then each of these integers can itself be represented either using the standard binary encoding of integers or using Chinese-Remaindering relative to the factors of q. We usually use the standard binary encoding for the coefficient representation and Chinese-Remaindering for the evaluation representation. (Hence the latter representation is really a *double CRT* representation, relative to both the polynomial factors of $\Phi_m(X)$ and the integer factors of q.)

2.1 BGV-type Cryptosystems

Our implementation uses a variant of the BGV cryptosystem due to Gentry, Halevi and Smart, specifically the one described in [11, Appendix D] (in the full version). In this cryptosystem both ciphertexts and secret keys are vectors over the polynomial ring $\mathbb{A}$, and the native plaintext space is the space of binary polynomials $\mathbb{A}_2$. (More generally it could be $\mathbb{A}_p$ for some fixed $p \geq 2$, but in our case we will always use $\mathbb{A}_2$.)

At any point during the homomorphic evaluation there is some "current integer modulus q" and "current secret key $\mathbf{s}$", that change from time to time. A ciphertext $\mathbf{c}$ is decrypted using the current secret key $\mathbf{s}$ by taking inner product over $\mathbb{A}_q$ (with q the current modulus) and then reducing the result modulo 2 *in coefficient representation*. Namely, the decryption formula is

$$a \leftarrow [\ \underbrace{[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_q}_{\text{noise}}\ ]_2 \ . \tag{1}$$

The polynomial $[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_q$ is called the "noise" in the ciphertext $\mathbf{c}$. Informally, $\mathbf{c}$ is a *valid ciphertext* with respect to secret key $\mathbf{s}$ and modulus q if this noise has "sufficiently small norm" relative to q. The meaning of "sufficiently small norm" is whatever is needed to ensure that the noise does not wrap around q when performing homomorphic operations, in our implementation we keep the norm of the noise always below some pre-set bound (which is determined in the full version of the paper).

Following [16, 11], the specific norm that we use to evaluate the magnitude of the noise is the "canonical embedding norm reduced mod q", specifically we use the conventions as described in [11, Appendix D] (in the full version). This is useful to get smaller parameters, but for the purpose of presentation the reader can think of the norm as the Euclidean norm of the noise in coefficient representation. More details are given in the Appendices. We refer to the norm of the noise as *the noise magnitude*.

The central feature of BGV-type cryptosystems is that the current secret key and modulus evolve as we apply operations to ciphertexts. We apply five different operations to ciphertexts during homomorphic evaluation. Three of them — addition, multiplication, and automorphism — are "semantic operations" that we use to evolve the plaintext data which is encrypted under those ciphertexts. The other two operations — key-switching and modulus-switching — are used for "maintenance": These operations do not change the plaintext at all, they only change the current key or modulus (respectively), and they are mainly used to control the complexity of the evaluation. Below we briefly describe each of these five operations on a high level. For reasons of space, key generation and encryption are described in the full version of the paper, with even more details being provided in [11, Appendix D].

Addition. Homomorphic addition of two ciphertext vectors with respect to the same secret key and modulus q is done just by adding the vectors over $\mathbb{A}_q$. If the two arguments were encrypting the plaintext polynomials $a_1, a_2 \in \mathbb{A}_2$ then the sum will be an encryption of $a_1 + a_2 \in \mathbb{A}_2$. This operation has no effect on the current modulus or key, and the norm of the noise is at most the sum of norms from the noise in the two arguments.

Multiplication. Homomorphic multiplication is done via tensor product over $\mathbb{A}_q$. In principle, if the two arguments have dimension n over $\mathbb{A}_q$ then the product ciphertext has dimension n^2, each entry in the output computed as the product of one entry from the first argument and one entry from the second.[5]

This operation does not change the current modulus, but it changes the current key: If the two input ciphertexts are valid with respect to the dimension-n secret key vector $\mathbf{s}$, encrypting the plaintext polynomials $a_1, a_2 \in \mathbb{A}_2$, then the output is valid with respect to the dimension-n^2 secret key $\mathbf{s}'$ which is the tensor product of $\mathbf{s}$ with itself, and it encrypts the polynomial $a_1 \cdot a_2 \in \mathbb{A}_2$. The norm of the noise in the product ciphertext can be bounded in terms of the product of norms of the noise in the two arguments. For our choice of norm function, the norm of the product is no larger than the product of the norms of the two arguments.

Key Switching. The public key of BGV-type cryptosystems includes additional components to enable converting a valid ciphertext with respect to one key into a valid

[5] It was shown in [5] that over polynomial rings this operation can be implemented while increasing the dimension only to $2n - 1$ rather than to n^2.

ciphertext encrypting the same plaintext with respect to another key. For example, this is used to convert the product ciphertext which is valid with respect to a high-dimension key back to a ciphertext with respect to the original low-dimension key.

To allow conversion from dimension-n' key $\mathbf{s}'$ to dimension-n key $\mathbf{s}$ (both with respect to the same modulus q), we include in the public key a matrix $W = W[\mathbf{s}' \to \mathbf{s}]$ over $\mathbb{A}_q$, where the i'th column of W is roughly an encryption of the i'th entry of $\mathbf{s}'$ with respect to $\mathbf{s}$ (and the current modulus). Then given a valid ciphertext $\mathbf{c}'$ with respect to $\mathbf{s}'$, we roughly compute $\mathbf{c} = W \cdot \mathbf{c}'$ to get a valid ciphertext with respect to $\mathbf{s}$.

In some more detail, the BGV key switching transformation first ensures that the norm of the ciphertext $\mathbf{c}'$ itself is sufficiently low with respect to q. In [3] this was done by working with the binary encoding of $\mathbf{c}'$, and one of our main optimization in this work is a different method for achieving the same goal (cf. Section 3.1). Then, if the i'th entry in $\mathbf{s}'$ is $\mathfrak{s}'_i \in \mathbb{A}$ (with norm smaller than q), then the i'th column of $W[\mathbf{s}' \to \mathbf{s}]$ is an n-vector $\mathbf{w}_i$ such that $[\langle \mathbf{w}_i, \mathbf{s} \rangle \bmod \Phi_m(X)]_q = 2e_i + \mathfrak{s}'_i$ for a low-norm polynomial $e_i \in \mathbb{A}$. Denoting $\mathbf{e} = (e_1, \ldots, e_{n'})$, this means that we have $\mathbf{s}W = \mathbf{s}' + 2\mathbf{e}$ over $\mathbb{A}_q$. For any ciphertext vector $\mathbf{c}'$, setting $\mathbf{c} = W \cdot \mathbf{c}' \in \mathbb{A}_q$ we get the equation

$$[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_q = [\mathbf{s}W\mathbf{c}' \bmod \Phi_m(X)]_q = [\langle \mathbf{c}', \mathbf{s}' \rangle + 2\langle \mathbf{c}', \mathbf{e} \rangle \bmod \Phi_m(X)]_q$$

Since $\mathbf{c}'$, $\mathbf{e}$, and $[\langle \mathbf{c}', \mathbf{s}' \rangle \bmod \Phi_m(X)]_q$ all have low norm relative to q, then the addition on the right-hand side does not cause a wrap around q, hence we get $[[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_q]_2 = [[\langle \mathbf{c}', \mathbf{s}' \rangle \bmod \Phi_m(X)]_q]_2$, as needed. The key-switching operation changes the current secret key from $\mathbf{s}'$ to $\mathbf{s}$, and does not change the current modulus. The norm of the noise is increased by at most an additive factor of $2\| \langle \mathbf{c}', \mathbf{e} \rangle \|$.

Modulus Switching. The modulus switching operation is intended to reduce the norm of the noise, to compensate for the noise increase that results from all the other operations. To convert a ciphertext $\mathbf{c}$ with respect to secret key $\mathbf{s}$ and modulus q into a ciphertext $\mathbf{c}'$ encrypting the same thing with respect to the same secret key but modulus q', we roughly just scale $\mathbf{c}$ by a factor q'/q (thus getting a fractional ciphertext), then round appropriately to get back an integer ciphertext. Specifically $\mathbf{c}'$ is a ciphertext vector satisfying (a) $\mathbf{c}' = \mathbf{c} \pmod 2$, and (b) the "rounding error term" $\tau \stackrel{\text{def}}{=} \mathbf{c}' - (q'/q)\mathbf{c}$ has low norm. Converting $\mathbf{c}$ to $\mathbf{c}'$ is easy in coefficient representation, and one of our optimizations is a method for doing the same in evaluation representation (cf. Section 3.2) This operation leaves the current key $\mathbf{s}$ unchanged, changes the current modulus from q to q', and the norm of the noise is changed as $\|n'\| \leq (q'/q)\|n\| + \|\tau \cdot s\|$. Note that if the key $\mathbf{s}$ has low norm and q' is sufficiently smaller than q, then the noise magnitude decreases by this operation.

A BGV-type cryptosystem has a chain of moduli, $q_0 < q_1 \cdots < q_{L-1}$, where fresh ciphertexts are with respect to the largest modulus q_{L-1}. During homomorphic evaluation every time the (estimated) noise grows too large we apply modulus switching from q_i to q_{i-1} in order to decrease it back. Eventually we get ciphertexts with respect to the smallest modulus q_0, and we cannot compute on them anymore (except by using bootstrapping).

Automorphisms. In addition to adding and multiplying polynomials, another useful operation is converting the polynomial $a(X) \in \mathbb{A}$ to $a^{(i)}(X) \stackrel{\text{def}}{=} a(X^i) \bmod \Phi_m(X)$.

Denoting by κ_i the transformation $\kappa_i : a \mapsto a^{(i)}$, it is a standard fact that the set of transformations $\{\kappa_i : i \in (\mathbb{Z}/m\mathbb{Z})^*\}$ forms a group under composition (which is the Galois group $\mathcal{G}\mathsf{al}(\mathbb{Q}(\zeta_m)/\mathbb{Q})$), and this group is isomorphic to $(\mathbb{Z}/m\mathbb{Z})^*$. In [3, 11] it was shown that applying the transformations κ_i to the plaintext polynomials is very useful, some more examples of its use can be found in our Section 4.

Denoting by $\mathbf{c}^{(i)}$, $\mathbf{s}^{(i)}$ the vector obtained by applying κ_i to each entry in $\mathbf{c}, \mathbf{s}$, respectively, it was shown in [3, 11] that if $\mathbf{s}$ is a valid ciphertext encrypting a with respect to key $\mathbf{s}$ and modulus q, then $\mathbf{c}^{(i)}$ is a valid ciphertext encrypting $a^{(i)}$ with respect to key $\mathbf{s}^{(i)}$ and the same modulus q. Moreover the norm of noise remains the same under this operation. We remark that we can apply key-switching to $\mathbf{c}^{(i)}$ in order to get an encryption of $a^{(i)}$ with respect to the original key $\mathbf{s}$.

2.2 Computing on Packed Ciphertexts

Smart and Vercauteren observed [19, 20] that the plaintext space $\mathbb{A}_2$ can be viewed as a vector of "plaintext slots", by an application the polynomial Chinese Remainder Theorem. Specifically, if the ring polynomial $\Phi_m(X)$ factors modulo 2 into a product of irreducible factors $\Phi_m(X) = \prod_{j=0}^{\ell-1} F_j(X) \pmod 2$, then a plaintext polynomial $a(X) \in \mathbb{A}_2$ can be viewed as encoding ℓ different small polynomials, $a_j = a \bmod F_j$. Just like for integer Chinese Remaindering, addition and multiplication in $\mathbb{A}_2$ correspond to element-wise addition and multiplication of the vectors of slots.

The effect of the automorphisms is a little more involved. When i is a power of two then the transformations $\kappa_i : a \mapsto a^{(i)}$ is just applied to each slot separately. When i is not a power of two the transformation κ_i has the effect of roughly shifting the values between the different slots. For example, for some parameters we could get a cyclic shift of the vector of slots: If a encodes the vector $(a_0, a_1, \ldots, a_{\ell-1})$, then $\kappa_i(a)$ (for some i) could encode the vector $(a_{\ell-1}, a_0, \ldots, a_{\ell-2})$. This was used in [11] to devise efficient procedures for applying arbitrary permutations to the plaintext slots.

We note that the values in the plaintext slots are not just bits, rather they are polynomials modulo the irreducible F_j's, so they can be used to represents elements in extension fields GF(2^d). In particular, in some of our AES implementations we used the plaintext slots to hold elements of GF(2^8), and encrypt one byte of the AES state in each slot. Then we can use an adaption of the techniques from [11] to permute the slots when performing the AES row-shift and column-mix.

3 General-Purpose Optimizations

Below we summarize our optimizations that are not tied directly to the AES circuit and can be used also in homomorphic evaluation of other circuits. Underlying many of these optimizations is our choice of keeping ciphertext and key-switching matrices in evaluation (double-CRT) representation. Our chain of moduli is defined via a set of primes of roughly the same size, $p_0, \ldots, p_{L-1}$, all chosen such that $\mathbb{Z}/p_i\mathbb{Z}$ has a m'th roots of unity. (In other words, $m|p_i - 1$ for all i.) For $i = 0, \ldots, L-1$ we then define our i'th modulus as $q_i = \prod_{j=0}^{i} p_i$. The primes p_0 and p_{L-1} are special (p_0 is chosen to ensure decryption works, and p_{L-1} is chosen to control noise immediately

after encryption), however all other primes p_i are of size $2^{17} \leq p_i \leq 2^{20}$ if $L < 100$, see the full version for further details.

In the t-th level of the scheme we have ciphertexts consisting of elements in $\mathbb{A}_{q_t}$ (i.e., polynomials modulo $(\Phi_m(X), q_t)$). We represent an element $c \in \mathbb{A}_{q_t}$ by a $\phi(m) \times (t+1)$ "matrix" of its evaluations at the primitive m-th roots of unity modulo the primes $p_0, \ldots, p_t$. Computing this representation from the coefficient representation of c involves reducing c modulo the p_i's and then $t+1$ invocations of the FFT algorithm, modulo each of the p_i (picking only the FFT coefficients corresponding to $(\mathbb{Z}/m\mathbb{Z})^*$). To convert back to coefficient representation we invoke the inverse FFT algorithm $t+1$ times, each time padding the $\phi(m)$-vector of evaluation point with $m - \phi(m)$ zeros (for the evaluations at the non-primitive roots of unity). This yields the coefficients of $t+1$ polynomials modulo $(X^m - 1, p_i)$ for $i = 0, \ldots, t$, we then reduce each of these polynomials modulo $(\Phi_m(X), p_i)$ and apply Chinese Remainder interpolation. We stress that we try to perform these transformations as rarely as we can.

3.1 A New Variant of Key Switching

As described in Section 2, the key-switching transformation introduces an additive factor of $2\langle \mathbf{c}', \mathbf{e} \rangle$ in the noise, where $\mathbf{c}'$ is the input ciphertext and $\mathbf{e}$ is the noise component in the key-switching matrix. To keep the noise magnitude below the modulus q, it seems that we need to ensure that the ciphertext $\mathbf{c}'$ itself has low norm. In BGV [3] this was done by representing $\mathbf{c}'$ as a fixed linear combination of small vectors, i.e. $\mathbf{c}' = \sum_i 2^i \mathbf{c}'_i$ with $\mathbf{c}'_i$ the vector of i'th bits in $\mathbf{c}'$. Considering the high-dimension ciphertext $\mathbf{c}^* = (\mathbf{c}'_0 | \mathbf{c}'_1 | \mathbf{c}'_2 | \cdots)$ and secret key $\mathbf{s}^* = (\mathbf{s}' | 2\mathbf{s}' | 4\mathbf{s}' | \cdots)$, we note that we have $\langle \mathbf{c}^*, \mathbf{s}^* \rangle = \langle \mathbf{c}', \mathbf{s}' \rangle$, and $\mathbf{c}^*$ has low norm (since it consists of 0-1 polynomials). BGV therefore included in the public key the matrix $W = W[\mathbf{s}^* \to \mathbf{s}]$ (rather than $W[\mathbf{s}' \to \mathbf{s}]$), and had the key-switching transformation computes $\mathbf{c}^*$ from $\mathbf{c}'$ and sets $\mathbf{c} = W \cdot \mathbf{c}^*$.

When implementing key-switching, there are two drawbacks to the above approach. First, this increases the dimension (and hence the size) of the key switching matrix. This drawback is fatal when evaluating deep circuits, since having enough memory to keep the key-switching matrices turns out to be the limiting factor in our ability to evaluate these deep circuits. In addition, for this key-switching we must first convert $\mathbf{c}'$ to coefficient representation (in order to compute the $\mathbf{c}'_i$'s), then convert each of the $\mathbf{c}'_i$'s back to evaluation representation before multiplying by the key-switching matrix. In level t of the circuit, this seem to require $\Omega(t \log q_t)$ FFTs.

In this work we propose a different variant: Rather than manipulating $\mathbf{c}'$ to decrease its norm, we instead temporarily increase the modulus q. We recall that for a valid ciphertext $\mathbf{c}'$, encrypting plaintext a with respect to $\mathbf{s}'$ and q, we have the equality $\langle \mathbf{c}', \mathbf{s}' \rangle = 2e' + a$ over A_q, for a low-norm polynomial e'. This equality, we note, implies that for every odd integer p we have the equality $\langle \mathbf{c}', p\mathbf{s}' \rangle = 2e'' + a$, *holding over* A_{pq}, for the "low-norm" polynomial e'' (namely $e'' = p \cdot e' + \frac{p-1}{2} a$). Clearly, when considered relative to secret key $p\mathbf{s}$ and modulus pq, the noise in $\mathbf{c}'$ is p times larger than it was relative to $\mathbf{s}$ and q. However, since the modulus is also p times larger, we maintain that the noise has norm sufficiently smaller than the modulus. In other words, $\mathbf{c}'$ is still a valid ciphertext that encrypts the same plaintext a with respect to secret key

$p\mathbf{s}$ and modulus pq. By taking p large enough, we can ensure that the norm of $\mathbf{c}'$ (which is independent of p) is sufficiently small relative to the modulus pq.

We therefore include in the public key a matrix $W = W[p\mathbf{s}' \to \mathbf{s}]$ modulo pq for a large enough odd integer p. (Specifically we need $p \approx q\sqrt{m}$.) Given a ciphertext $\mathbf{c}'$, valid with respect to $\mathbf{s}$ and q, we apply the key-switching transformation simply by setting $\mathbf{c} = W \cdot \mathbf{c}'$ over $\mathbb{A}_{pq}$. The additive noise term $\langle \mathbf{c}', \mathbf{e} \rangle$ that we get is now small enough relative to our large modulus pq, thus the resulting ciphertext $\mathbf{c}$ is valid with respect to $\mathbf{s}$ and pq. We can now switch the modulus back to q (using our modulus switching routine), hence getting a valid ciphertext with respect to $\mathbf{s}$ and q.

We note that even though we no longer break $\mathbf{c}'$ into its binary encoding, it seems that we still need to recover it in coefficient representation in order to compute the evaluations of $\mathbf{c}' \bmod p$. However, since we do not increase the dimension of the ciphertext vector, this procedure requires only $O(t)$ FFTs in level t (vs. $O(t \log q_t) = O(t^2)$ for the original BGV variant). Also, the size of the key-switching matrix is reduced by roughly the same factor of $\log q_t$.

Our new variant comes with a price tag, however: We use key-switching matrices relative to a larger modulus, but still need the noise term in this matrix to be small. This means that the LWE problem underlying this key-switching matrix has larger ratio of modulus/noise, implying that we need a larger dimension to get the same level of security than with the original BGV variant. In fact, since our modulus is more than squared (from q to pq with $p > q$), the dimension is increased by more than a factor of two. This translates to more than doubling of the key-switching matrix, partly negating the size and running time advantage that we get from this variant.

We comment that a hybrid of the two approaches could also be used: we can decrease the norm of $\mathbf{c}'$ only somewhat by breaking it into digits (as opposed to binary bits as in [3]), and then increase the modulus somewhat until it is large enough relative to the smaller norm of $\mathbf{c}'$. We speculate that the optimal setting in terms of runtime is found around $p \approx \sqrt{q}$, but so far did not try to explore this tradeoff.

3.2 Modulus Switching in Evaluation Representation

Given an element $c \in \mathbb{A}_{q_t}$ in evaluation (double-CRT) representation relative to $q_t = \prod_{j=0}^{t} p_j$, we want to modulus-switch to q_{t-1} – i.e., scale down by a factor of p_t; we call this operation $\mathsf{Scale}(c, q_t, q_{t-1})$ The output should be $c' \in \mathbb{A}$, represented via the same double-CRT format (with respect to $p_0, \ldots, p_{t-1}$), such that (a) $c' \equiv c \pmod 2$, and (b) the "rounding error term" $\tau = c' - (c/p_t)$ has a very low norm. As p_t is odd, we can equivalently require that the element $c^\dagger \stackrel{\text{def}}{=} p_t \cdot c'$ satisfy

(i) $c^\dagger$ is divisible by p_t,
(ii) $c^\dagger \equiv c \pmod 2$, and
(iii) $c^\dagger - c$ (which is equal to $p_t \cdot \tau$) has low norm.

Rather than computing c' directly, we will first compute $c^\dagger$ and then set $c' \leftarrow c^\dagger / p_t$. Observe that once we compute $c^\dagger$ in double-CRT format, it is easy to output also c' in double-CRT format: given the evaluations for $c^\dagger$ modulo p_j $(j < t)$, simply multiply them by $p_t^{-1} \bmod p_j$. The algorithm to output $c^\dagger$ in double-CRT format is as follows:

1. Set $\bar{c}$ to be the coefficient representation of $c \bmod p_t$. (Computing this requires a single "small FFT" modulo the prime p_t.)
2. Add or subtract p_t from every odd coefficient of $\bar{c}$, thus obtaining a polynomial δ with coefficients in $(-p_t, p_t]$ such that $\delta \equiv \bar{c} \equiv c \pmod{p_t}$ and $\delta \equiv 0 \pmod{2}$.
3. Set $c^\dagger = c - \delta$, and output it in double-CRT representation.
   Since we already have c in double-CRT representation, we only need the double-CRT representation of δ, which requires t more "small FFTs" modulo the p_j's.

As all the coefficients of $c^\dagger$ are within p_t of those of c, the "rounding error term" $\tau = (c^\dagger - c)/p_t$ has coefficients of magnitude at most one, hence it has low norm.

The procedure above uses $t + 1$ small FFTs in total. This should be compared to the naive method of just converting everything to coefficient representation modulo the primes ($t + 1$ FFTs), CRT-interpolating the coefficients, dividing and rounding appropriately the large integers (of size $\approx q_t$), CRT-decomposing the coefficients, and then converting back to evaluation representation ($t + 1$ more FFTs). The above approach makes explicit use of the fact that we are working in a plaintext space modulo 2; in the full version we present a technique which works when the plaintext space is defined modulo a larger modulus.

3.3 Dynamic Noise Management

As described in the literature, BGV-type cryptosystems tacitly assume that each homomorphic operation operation is followed a modulus switch to reduce the noise magnitude. In our implementation, however, we attach to each ciphertext an estimate of the noise magnitude in that ciphertext, and use these estimates to decide dynamically when a modulus switch must be performed.

Each modulus switch consumes a level, and hence a goal is to reduce, over a computation, the number of levels consumed. By paying particular attention to the parameters of the scheme, and by carefully analyzing how various operations affect the noise, we are able to control the noise much more carefully than in prior work. In particular, we note that modulus-switching is really only necessary just prior to multiplication (when the noise magnitude is about to get squared), in other times it is acceptable to keep the ciphertexts at a higher level (with higher noise).

3.4 Randomized Multiplication by Constants

Our implementation of the AES round function uses just a few multiplication operations (only seven per byte!), but it requires a relatively large number of multiplications of encrypted bytes by constants. Hence it becomes important to try and squeeze down the increase in noise when multiplying by a constant. To that end, we encode a constant polynomial in $\mathbb{A}_2$ as a polynomial with coefficients in $\{-1, 0, 1\}$ rather than in $\{0, 1\}$. Namely, we have a procedure $\mathsf{Randomize}(\alpha)$ that takes a polynomial $\alpha \in \mathbb{A}_2$ and replaces each non-zero coefficients with a coefficients chosen uniformly from $\{-1, 1\}$. By Chernoff bound, we expect that for α with h nonzero coefficients, the canonical embedding norm of $\mathsf{Randomize}(\alpha)$ to be bounded by $O(\sqrt{h})$ with high probability (assuming that h is large enough for the bound to kick in). This yields a better bound

on the noise increase than the trivial bound of h that we would get if we just multiply by α itself. (In the full version we present a heuristic argument that we use to bound the noise, which yields the same asymptotic bounds but slightly better constants.)

4 Homomorphic Evaluation of AES

Next we describe our homomorphic implementation of AES-128. We implemented three distinct implementation possibilities; we first describe the "packed implementation", in which the entire AES state is packed in just one ciphertext. Two other implementations (of byte-slice and bit-slice AES) are described later in Section 4.2. The "packed" implementation uses the least amount of memory (which turns out to be the main constraint in our implementation), and also the fastest running time for a single evaluation. The other implementation choices allow more SIMD parallelism, on the other hand, so they can give better amortized running time when evaluating AES on many blocks in parallel.

A Brief Overview of AES. The AES-128 cipher consists of ten applications of the same keyed round function (with different round keys). The round function operates on a 4×4 matrix of bytes, which are sometimes considered as element of $\mathbb{F}_{2^8}$. The basic operations that are performed during the round function are AddKey, SubBytes, ShiftRows, MixColumns. The AddKey is simply an XOR operation of the current state with 16 bytes of key; the SubBytes operation consists of an inversion in the field $\mathbb{F}_{2^8}$ followed by a fixed $\mathbb{F}_2$-linear map on the bits of the element (relative to a fixed polynomial representation of $\mathbb{F}_{2^8}$); the ShiftRows rotates the entries in the row i of the 4×4 matrix by $i-1$ places to the left; finally the MixColumns operations pre-multiplies the state matrix by a fixed 4×4 matrix.

Our Packed Representation of the AES state. For our implementation we chose the native plaintext space of our homomorphic encryption so as to support operations on the finite field $\mathbb{F}_{2^8}$. To this end we choose our ring polynomial as $\Phi_m(X)$ that factors modulo 2 into degree-d irreducible polynomials such that $8|d$. (In other words, the smallest integer d such that $m|(2^d-1)$ is divisible by 8.) This means that our plaintext slots can hold elements of $\mathbb{F}_{2^d}$, and in particular we can use them to hold elements of $\mathbb{F}_{2^8}$ which is a sub-field of $\mathbb{F}_{2^d}$. Since we have $\ell = \phi(m)/d$ plaintext slots in each ciphertext, we can represent upto $\lfloor \ell/16 \rfloor$ complete AES state matrices per ciphertext.

Moreover, we choose our parameter m so that there exists an element $g \in \mathbb{Z}_m^*$ that has order 16 in both $\mathbb{Z}_m^*$ and the quotient group $\mathbb{Z}_m^*/\langle 2 \rangle$. This condition means that if we put 16 plaintext bytes in slots $t, tg, tg^2, tg^3, \ldots$ (for some $t \in \mathbb{Z}_m^*$), then the conjugation operation $X \mapsto X^g$ implements a cyclic right shift over these sixteen plaintext bytes.

In the computation of the AES round function we use several constants. Some constants are used in the S-box lookup phase to implement the AES bit-affine transformation, these are denoted γ and γ_{2^j} for $j = 0, \ldots, 7$. In the row-shift/col-mix part we use a constant $\mathsf{C_{slct}}$ that has 1 in slots corresponding to $t \cdot g^i$ for $i = 0, 4, 8, 12$, and 0 in all the other slots of the form $t \cdot g^i$. (Here slot t is where we put the first AES byte.) We also use 'X' to denote the constant that has the element X in all the slots.

4.1 Homomorphic Evaluation of the Basic Operations

We now examine each AES operation in turn, and describe how it is implemented homomorphically. For each operation we denote the plaintext polynomial underlying a given input ciphertext $\mathfrak{c}$ by a, and the corresponding content of the ℓ plaintext slots are denoted as an ℓ-vector $(\alpha_i)_{i=1}^{\ell}$, with each $\alpha_i \in \mathbb{F}_{2^8}$.

AddKey and SubBytes The AddKey is just a simple addition of ciphertexts, which yields a 4×4 matrix of bytes in the input to the SubBytes operation. We place these 16 bytes in plaintext slots tg^i for $i = 0, 1, \ldots, 15$, using column-ordering to decide which byte goes in what slot, namely we have

$$a \approx [\alpha_{00}\alpha_{10}\alpha_{20}\alpha_{30}\alpha_{01}\alpha_{11}\alpha_{21}\alpha_{31}\alpha_{02}\alpha_{12}\alpha_{22}\alpha_{32}\alpha_{03}\alpha_{13}\alpha_{23}\alpha_{33}],$$

encrypting the input plaintext matrix

$$A = \big(\alpha_{ij}\big)_{i,j} \quad = \quad \begin{pmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{pmatrix}.$$

During S-box lookup, each plaintext byte α_{ij} should be replaced by $\beta_{ij} = S(\alpha_{ij})$, where $S(\cdot)$ is a fixed permutation on the bytes. Specifically, $S(x)$ is obtained by first computing $y = x^{-1}$ in $\mathbb{F}_{2^8}$ (with 0 mapped to 0), then applying a bitwise affine transformation $z = T(y)$ where elements in $\mathbb{F}_{2^8}$ are treated as bit strings with representation polynomial $G(X) = x^8 + x^4 + x^3 + x + 1$.

We implement $\mathbb{F}_{2^8}$ inversion followed by the $\mathbb{F}_2$ affine transformation using the Frobenius automorphisms, $X \longrightarrow X^{2^j}$. Recall that for a power of two $k = 2^j$, the transformation $\kappa_k(a(X)) = (a(X^k) \bmod \Phi_m(X))$ is applied separately to each slot, hence we can use it to transform the vector $(\alpha_i)_{i=1}^{\ell}$ into $(\alpha_i^k)_{i=1}^{\ell}$. We note that applying the Frobenius automorphisms to ciphertexts has almost no influence on the noise magnitude, and hence it does not consume any levels.[6]

Inversion over $\mathbb{F}_{2^8}$ is done using essentially the same procedure as Algorithm 2 from [18] for computing $\beta = \alpha^{-1} = \alpha^{254}$. This procedure takes only three Frobenius automorphisms and four multiplications, arranged in a depth-3 circuit (see details below.) To apply the AES $\mathbb{F}_2$ affine transformation, we use the fact that any $\mathbb{F}_2$ affine transformation can be computed as a $\mathbb{F}_{2^8}$ affine transformation over the conjugates. Thus there are constants $\gamma_0, \gamma_1, \ldots, \gamma_7, \delta \in \mathbb{F}_{2^8}$ such that the AES affine transformation $T_{\mathrm{AES}}(\cdot)$ can be expressed as $T_{\mathrm{AES}}(\beta) = \delta + \sum_{j=0}^{7} \gamma_j \cdot \beta^{2^j}$ over $\mathbb{F}_{2^8}$. We therefore again apply the Frobenius automorphisms to compute eight ciphertexts encrypting the polynomials $\kappa_k(b)$ for $k = 1, 2, 4, \ldots, 128$, and take the appropriate linear combination (with coefficients the γ_j's) to get an encryption of the vector $(T_{\mathrm{AES}}(\alpha_i^{-1}))_{i=1}^{\ell}$. For our parameters, a multiplication-by-constant operation consumes roughly half a level in terms of added noise.

[6] It does increase the noise magnitude somewhat, because we need to do key switching after these automorphisms. But this is only a small influence, and we will ignore it here.

One subtle implementation detail to note here, is that although our plaintext slots all hold elements of the same field $\mathbb{F}_{2^8}$, they hold these elements with respect to different polynomial encodings. The AES affine transformation, on the other hand, is defined with respect to one particular fixed polynomial encoding. This means that we must implement in the i'th slot not the affine transformation $T_{\mathrm{AES}}(\cdot)$ itself but rather the projection of this transformation onto the appropriate polynomial encoding: When we take the affine transformation of the eight ciphertexts encrypting $b_j = \kappa_{2^j}(b)$, we therefore multiply the encryption of b_j not by a constant that has γ_j in all the slots, but rather by a constant that has in slot i the projection of γ_j to the polynomial encoding of slot i.

Below we provide a pseudo-code description of our S-box lookup implementation, together with an approximation of the levels that are consumed by these operations. (These approximations are somewhat under-estimates, however.)

	Level	
Input: ciphertext $\mathfrak{c}$	t	
// Compute $\mathfrak{c}_{254} = \mathfrak{c}^{-1}$		
1. $\mathfrak{c}_2 \leftarrow \mathfrak{c} \gg 2$	t	// Frobenius $X \mapsto X^2$
2. $\mathfrak{c}_3 \leftarrow \mathfrak{c} \times \mathfrak{c}_2$	$t+1$	// Multiplication
3. $\mathfrak{c}_{12} \leftarrow \mathfrak{c}_3 \gg 4$	$t+1$	// Frobenius $X \mapsto X^4$
4. $\mathfrak{c}_{14} \leftarrow \mathfrak{c}_{12} \times \mathfrak{c}_2$	$t+2$	// Multiplication
5. $\mathfrak{c}_{15} \leftarrow \mathfrak{c}_{12} \times \mathfrak{c}_3$	$t+2$	// Multiplication
6. $\mathfrak{c}_{240} \leftarrow \mathfrak{c}_{15} \gg 16$	$t+2$	// Frobenius $X \mapsto X^{16}$
7. $\mathfrak{c}_{254} \leftarrow \mathfrak{c}_{240} \times \mathfrak{c}_{14}$	$t+3$	// Multiplication
// Affine transformation over $\mathbb{F}_2$		
8. $\mathfrak{c}'_{2^j} \leftarrow \mathfrak{c}_{254} \gg 2^j$ for $j = 0, 1, 2, \ldots, 7$	$t+3$	// Frobenius $X \mapsto X^{2^j}$
9. $\mathfrak{c}'' \leftarrow \gamma + \sum_{j=0}^{7} \gamma_j \times \mathfrak{c}'_{2^j}$	$t+3.5$	// Linear combination over $\mathbb{F}_{2^8}$

ShiftRows and MixColumns As commonly done, we interleave the ShiftRows/MixColumns operations, viewing both as a single linear transformation over vectors from $(\mathbb{F}_{2^8})^{16}$. As mentioned above, by a careful choice of the parameter m and the placement of the AES state bytes in our plaintext slots, we can implement a rotation-by-i of the rows of the AES matrix as a single automorphism operations $X \mapsto X^{g^i}$ (for some element $g \in (\mathbb{Z}/m\mathbb{Z})^*$). Given the ciphertext $\mathfrak{c}''$ after the SubBytes step, we use these operations (in conjunction with ℓ-SELECT operations, as described in [11]) to compute four ciphertexts corresponding to the appropriate permutations of the 16 bytes (in each of the $\ell/16$ different input blocks). These four ciphertexts are combined via a linear operation (with coefficients $1, X$, and $(1+X)$) to obtain the final result of this round function. Below is a pseudo-code of this implementation and an approximation for the levels that it consumes (starting from $t - 3.5$). We note that the permutations are implemented using automorphisms and multiplication by constant, thus we expect them to consume roughly 1/2 level.

	Level	
Input: ciphertext $\mathfrak{c}''$	$t + 3.5$	
10. $\mathfrak{c}_j^* \leftarrow \pi_j(\mathfrak{c}'')$ for $j = 1, 2, 3, 4$	$t + 4.0$	// Permutations
11. Output $X \cdot \mathfrak{c}_1^* + (X + 1) \cdot \mathfrak{c}_2^* + \mathfrak{c}_3^* + \mathfrak{c}_4^*$	$t + 4.5$	// Linear combination

The Cost of One Round Function The above description yields an estimate of 5 levels for implementing one round function. This is however, an underestimate. The actual number of levels depends on details such as how sparse the scalars are with respect to the embedding via Φ_m in a given parameter set, as well as the accumulation of noise with respect to additions, Frobenius operations etc. Running over many different parameter sets we find the average number of levels per round for this method varies between 5.0 and 6.0.

We mention that the byte-slice and bit-slice implementations, given in Section 4.2 below, can consume less levels per round function, since they do not need to permute slots inside a single ciphertext. Specifically, for our byte-sliced implementation, we only need 4.5-5.0 levels per round on average. However, since we need to manipulate many more ciphertexts, the implementation takes much more time per evaluation and requires much more memory. On the other hand it offers wider parallelism, so yields better amortized time per block. Our bit-sliced implementation should theoretical consume the least number of levels (by purely counting multiplication gates), but the noise introduced by additions means the average number of levels consumed per round varies from 5.0 upto 10.0.

4.2 Byte- and Bit-Slice Implementations

In the byte sliced implementation we use sixteen distinct ciphertexts to represent a single state matrix. (But since each ciphertext can hold ℓ plaintext slots, then these 16 ciphertexts can hold the state of ℓ different AES blocks). In this representation there is no interaction between the slots, thus we operate with pure ℓ-fold SIMD operations. The AddKey and SubBytes steps are exactly as above (except applied to 16 ciphertexts rather than a single one). The permutations in the ShiftRows/MixColumns step are now "for free", but the scalar multiplication in MixColumns still consumes another level in the modulus chain.

Using the same estimates as above, we expect the number of levels per round to be roughly four (as opposed to the 4.5 of the packed implementation). In practice, again over many parameter sets, we find the average number of levels consumed per round is between 4.5 and 5.0.

For the bit sliced implementation we represent the entire round function as a binary circuit, and we use 128 distinct ciphertexts (one per bit of the state matrix). However each set of 128 ciphertexts is able to represent a total of ℓ distinct blocks. The main issue here is how to create a circuit for the round function which is as shallow, in terms of number of multiplication gates, as possible. Again the main issue is the SubBytes operation as all operations are essentially linear. To implement the SubBytes we used the "depth-16" circuit of Boyar and Peralta [1], which consumes four levels. The rest of the round function can be represented as a set of bit-additions, Thus, implementing

this method means that we consumes a minimum of four levels on computing an entire round function. However, the extensive additions within the Boyar–Peralta circuit mean that we actually end up consuming a lot more. On average this translates into actually consuming between 5.0 and 10.0 levels per round.

4.3 Performance Details

As remarked in the introduction, we implemented the above variant of evaluating AES homomorphically on a very large memory machine; namely a machine with 256 GB of RAM. Firstly parameters were selected, for details see the full version, to cope with 60 levels of computation, and a public/private key pair was generated; along with the key-switching data for multiplication operations and conjugation with-respect-to the Galois group.

As input to the actual computation was an AES plaintext block and the eleven round keys; each of which was encrypted using our homomorphic encryption scheme. Thus the input consisted of eleven packed ciphertexts. Producing the encrypted key schedule took around half an hour. To evaluate the entire ten rounds of AES took just over 36 hours; however each of our ciphertexts could hold 864 plaintext slots of elements in $\mathbb{F}_{2^8}$, thus we could have processed 54 such AES blocks in this time period. This would result in a throughput of around forty minutes per AES block.

We note that as the algorithm progressed the operations became faster. The first round of the AES function took 7 hours, whereas the penultimate round took 2 hours and the last round took 30 minutes. Recall, the last AES round is somewhat simpler as it does not involve a MixColumns operation.

Whilst our other two implementation choices (given in Section 4.2 below) may seem to yield better amortized per-block timing, the increase in memory requirements and data actually makes them less attractive when encrypting a single block. For example just encrypting the key schedule in the Byte-Sliced variant takes just under 5 hours (with 50 levels), with an entire encryption taking 65 hours (12 hours for the first round, with between 4 and 5 hours for both the penultimate and final rounds). This however equates to an amortized time of just over five minutes per block.

The Bit-Sliced variant requires over 150 hours to just encrypt the key schedule (with 60 levels), and evaluating a single round takes so long that our program is timed out before even a single round is evaluated.

5 Acknowledgments

We thank Jean-Sebastien Coron for pointing out to us the efficient implementation from [18] of the AES S-box lookup.

The first and second authors are sponsored by DARPA under agreement number FA8750-11-C-0096. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either

expressed or implied, of DARPA or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

The third author is sponsored by DARPA and AFRL under agreement number FA8750-11-2-0079. The same disclaimers as above apply. He is also supported by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II and via an ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant COED–EP/I03126X, and by a Royal Society Wolfson Merit Award. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the European Commission or EPSRC.

References

1. Joan Boyar and René Peralta. A depth-16 circuit for the AES S-box. Manuscript, `http://eprint.iacr.org/2011/332`, 2011.
2. Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. Manuscript, `http://eprint.iacr.org/2012/078`, 2012.
3. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science (ITCS'12)*, 2012. Available at `http://eprint.iacr.org/2011/277`.
4. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS'11*. IEEE Computer Society, 2011.
5. Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.
6. Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 487–504. Springer, 2011.
7. Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *Advances in Cryptology - EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 446–464. Springer, 2012.
8. Ivan Damgård and Marcel Keller. Secure multiparty aes. In *Proc. of Financial Cryptography 2010*, volume 6052 of *LNCS*, pages 367–374, 2010.
9. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.
10. Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2011.
11. Craig Gentry, Shai Halevi, and Nigel Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012. Full version at `http://eprint.iacr.org/2011/566`.
12. Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.
13. C. Orlandi J.B. Nielsen, P.S. Nordholt and S. Sheshank. A new approach to practical active-secure two-party computation. Manuscript, 2011.
14. Kristin Lauter, Michael Naehrig, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *CCSW*, pages 113–124. ACM, 2011.

15. Adriana Lòpez-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *STOC*. ACM, 2012.
16. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, 2010.
17. Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Steven C. Williams. Secure two-party computation is practical. In *Proc. ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267, 2009.
18. Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.
19. Nigel P. Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography - PKC'10*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer, 2010.
20. Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. Manuscript at http://eprint.iacr.org/2011/133, 2011.

A Other Optimizations

Some other optimizations that we encountered during our implementation work are discussed next. Not all of these optimizations are useful for our current implementation, but they may be useful in other contexts.

Three-way Multiplications. Sometime we need to multiply several ciphertexts together, and if their number is not a power of two then we do not have a complete binary tree of multiplications, which means that at some point in the process we will have three ciphertexts that we need to multiply together.

The standard way of implementing this 3-way multiplication is via two 2-argument multiplications, e.g., $x \cdot (y \cdot z)$. But it turns out that here it is better to use "raw multiplication" to multiply these three ciphertexts (as done in [5]), thus getting an "extended" ciphertext with four elements, then apply key-switching (and later modulus switching) to this ciphertext. This takes only six ring-multiplication operations (as opposed to eight according to the standard approach), three modulus switching (as opposed to four), and only one key switching (applied to this 4-element ciphertext) rather than two (which are applied to 3-element extended ciphertexts). All in all, this three-way multiplication takes roughly 1.5 times a standard two-element multiplication.

We stress that this technique is not useful for larger products, since for more than three multiplicands the noise begins to grow too large. But with only three multiplicands we get noise of roughly B^3 after the multiplication, which can be reduced to noise $\approx B$ by dropping two levels, and this is also what we get by using two standard two-element multiplications.

Commuting Automorphisms and Multiplications. Recalling that the automorphisms $X \mapsto X^i$ commute with the arithmetic operations, we note that some ordering of these operations can sometimes be better than others. For example, it may be better perform the multiplication-by-constant before the automorphism operation whenever possible. The reason is that if we perform the multiply-by-constant after the key-switching that

follows the automorphism, then added noise term due to that key-switching is multiplied by the same constant, thereby making the noise slightly larger. We note that to move the multiplication-by-constant before the automorphism, we need to multiply by a different constant.

Switching to higher-level moduli. We note that it may be better to perform automorphisms at a higher level, in order to make the added noise term due to key-switching small with respect to the modulus. On the other hand operations at high levels are more expensive than the same operations at a lower level. A good rule of thumb is to perform the automorphism operations one level above the lowest one. Namely, if the naive strategy that never switches to higher-level moduli would perform some Frobenius operation at level q_i, then we perform the key-switching following this Frobenius operation at level Q_{i+1}, and then switch back to level q_{i+1} (rather then using Q_i and q_i).

Commuting Addition and Modulus-switching. When we need to add many terms that were obtained from earlier operations (and their subsequent key-switching), it may be better to first add all of these terms relative to the large modulus Q_i before switching the sum down to the smaller q_i (as opposed to switching all the terms individually to q_i and then adding).

Reducing the number of key-switching matrices. When using many different automorphisms $\kappa_i : X \mapsto X^i$ we need to keep many different key-switching matrices in the public key, one for every value of i that we use. We can reduces this memory requirement, at the expense of taking longer to perform the automorphisms. We use the fact that the Galois group $\mathcal{G}\mathsf{al}$ that contains all the maps κ_i (which is isomorphic to $(\mathbb{Z}/m\mathbb{Z})^*$) is generated by a relatively small number of generators. (Specifically, for our choice of parameters the group $(\mathbb{Z}/m\mathbb{Z})^*$ has two or three generators.) It is therefore enough to store in the public key only the key-switching matrices corresponding to κ_{g_j}'s for these generators g_j of the group $\mathcal{G}\mathsf{al}$. Then in order to apply a map κ_i we express it as a product of the generators and apply these generators to get the effect of κ_i. (For example, if $i = g_1^2 \cdot g_2$ then we need to apply κ_{g_1} twice followed by a single application of κ_{g_2}.)