Armadillo: A Compilation Chain for Privacy Preserving Applications

Sergiu Carpov, Paul Dubrulle, Renaud Sirdey
CEA, LIST,
Point Courrier 172, 91191 Gif-sur-Yvette Cedex, France
{sergiu.carpov, paul.dubrulle, renaud.sirdey}@cea.fr

ABSTRACT

In this work we present Armadillo a compilation chain used for compiling applications written in a high-level language (C++) to work on encrypted data. The back-end of the compilation chain is based on homomorphic encryption. The tool-chain further automatically handle a huge amount of parallelism so as to mitigate the performance overhead of using homomorphic encryption.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, compilers*

General Terms

Security

Keywords

Homomorphic encryption; compilation chain; parallel execution.

1. INTRODUCTION

In parallel with the research work which has lead to dramatic improvements with respect to the computational overhead of homomorphic encryption (research which has been conducted for the most part within the cryptographic community) the compilation and parallelism community has also started to grow a strong interest to homomorphic encryption techniques as a new execution environment for computer programs with a highly promising practical relevance. In particular, it should be emphasized that a homomorphic encryption system mostly provides bit-level operators, hence intrinsically low level. Thus, making the connection between an algorithm written in a high-level programming language and such a low-level execution environment requires a sequence of non-trivial transformations, that is, a compiler. This even more so if it is required that the performance hit of homomorphic execution be mitigated, as much as possible, by means of optimised code generation and parallelism.

In this paper, we present Armadillo, a compiler and code generation environment aiming at bridging the gap between the level of abstraction of rather complex programs and algorithms, level of abstraction at which application designers are working, and the low-level formalism of homomorphic encryption. Armadillo aims at addressing the software engineering issues of cost-effectively writing programs for execution over encrypted data and automatically handling the large amount of parallelism required to do so with non prohibitive performances. Indeed, parallel programming and parallel program debugging are very difficult tasks for mainstream applicative programmers and generally must be automated in order not to induce large (and often underestimated) development costs. By providing an optimizing compiler and parallel runtime environment, Armadillo is thus a first attempt to address both facets of the software engineering cost issue of operationnally using homomorphic encryption. The first version of Armadillo is a pragmatic assembly of software building-blocks, some of them already existing and borrowed for seemingly unrelated fields, which demonstrates the possibility of building a full blown compiler environment for homomorphic encryption-based computing over encrypted data as well as of doing so at low software engineering cost and with decent performances on simple yet useful algorithms. In essence, Armadillo also provide a platform for the development and validation of more advanced homomorphic encryption code generation and optimization techniques. Furthermore, other cryptographic constructions (e.g. garbled circuits, functional encryption) can be integrated in Armadillo compilation chain.

2. FULLY HOMOMORPHIC ENCRYPTION

An *encryption scheme* describes the way of encrypting and decrypting plaintext messages such that finding which is the plaintext message from encrypted data (or ciphertext in what follows) is either very hard or even impossible. An encryption scheme is said to be *homomorphic* when some operations on plaintext messages can be done homomorphically, that is directly in the space of ciphertexts (without decrypting them). Addition and multiplication are two operations on plaintexts which can be done homomorphically, although other operations can be found in the literature. An encryption scheme is called *fully homomorphic* when both operations (addition and multiplication) are supported. A fully homomorphic encryption scheme allows to execute any boolean circuit directly on encrypted data. The first prac-

SCC'15, April 14-17 2015, Singapore, Singapore.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3447-1/15/04 ...$15.00.
http://dx.doi.org/10.1145/2732516.2732520.

tical (to some extent) fully homomorphic encryption (FHE) scheme was proposed by Gentry [12].

For security reasons a noise component is added to the ciphertext during the encryption. The noise component is a common characteristic for FHE schemes. Each new homomorphic operation applied on the ciphertexts increases the noise component in the resulting ciphertext. After a (predefined) number of homomorphic operations the noise is so large that no decryption is possible. Usually the noise growth induced by the addition operation is smaller than the noise growth induced by the multiplication operation. That is why many authors consider only the multiplicative depth[1] of evaluated circuits when FHE schemes are parametrized.

The ciphertext and plaintext in FHE schemes are either integer or polynomial ring elements. According to the literature the schemes over polynomial rings are asymptotically more efficient than the schemes based on integer rings [6]. If the ciphertexts sizes in both cases are roughly the same then the computations are heavier and the additional data (public and evaluation keys) have larger sizes for schemes over integer rings. In return the learning with errors (LWE) problem, on which is based the security of integer ring schemes, is better understood than the ring-LWE problem.

It can be considered that the plaintext space in FHE schemes are integer quotient rings $\mathbb{Z}_t$ ($t \geq 2$), in other words the plaintext are integers modulo t. We shall use modulo 2 plaintext in order to extend the set of supported high-level programming language instructions. Actually using FHE schemes encrypted data dependent control instructions are realisable only when the plaintext is binary. The majority of FHE schemes have a common set of operations: parameter generation, key generation, encryption/decryption, homomorphic addition and multiplication of ciphertexts, etc. Addition and multiplication operations can also be performed with one non-encrypted input, in this case the homomorphic operations are much lighter.

3. ARMADILLO COMPILATION CHAIN

In what follows, we consider fully homomorphic encryption schemes which support two operations: addition and multiplication modulo 2. These operations can be seen as XOR and AND logic gates. A program written in a high-level language is transformed into a boolean circuit in order to be able to execute it homomorphically. The Armadillo compilation chain provides an easy to use compiler which builds a privacy-preserving binary for an application written in a high-level language. The compilation chain is classically composed of 3 layers: a front-end, a middle-end and a back-end. The front-end transforms code written in the input language (C++) into its boolean circuit representation. The middle-end layer optimizes the boolean circuit produced by the front-end. The back-end constructs a binary which homomorphically executes the boolean circuit on encrypted data. In this work we limited ourselves only to shared memory architecture back-end (using C++/OpenMP language), but the software design of Armadillo allows to easily add supplementary backends. In the next sections we describe each layer of the compilation chain.

3.1 Front-end

The front-end aims to transform a C++ code into the form of a boolean circuit. This representation is build using a transparent programming interface. The boolean circuit and the programming interface which builds it is defined in what follows.

3.1.1 Boolean circuit

A boolean circuit is an acyclic directed graph $G = (V, E)$ with a set of vertices V and a set of edges E. The set of vertices can be split into 3 independent sub-sets:

- Vertices without a predecessor define circuit inputs. An input vertex can be either a boolean input variable or a boolean constant ("0" or "1" input vertices).
- Vertices representing a gate applying a basic boolean function to the values of its predecessors. The input degree of gate functions is either 1 or 2, defined by the function they represent.
- Vertices without a successor define circuit outputs. An output vertex has a single predecessor.

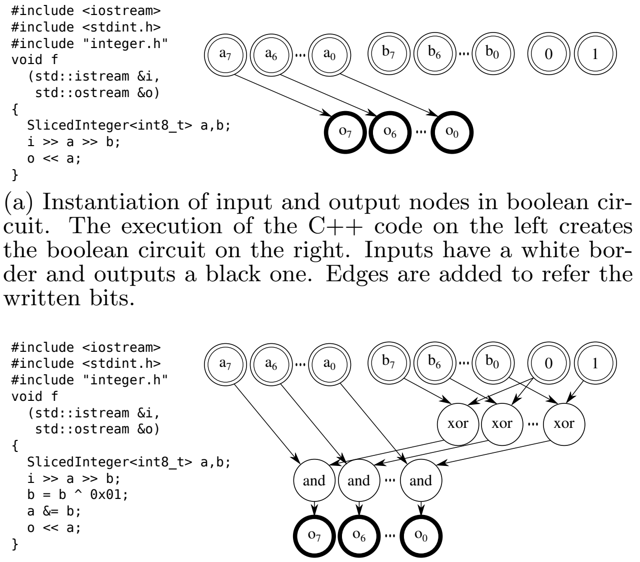
3.1.2 Generation

The generation phase builds the boolean circuit representing all the operations applied to the input bits during a normal execution. To achieve this, from an algorithm expressed in C++, we take advantage of the so-called *template* classes. We provide a class `SlicedInteger` whose instantiations represent encrypted variables used in the algorithm.

The template class of composite integers `SlicedInteger` encodes a collection of objects representing its bits, and thus splits integer operations in a bit-wise fashion. Splitting algorithms at bit level is not novel and has been used before [19, 9]. Adding two objects of this class is implemented as a standard adder with carry propagation. The template is instantiated with a basic integer type, which defines its size in bits. The bits are represented by objects of a `BitTracker` class, which tracks operations and records them under the form of a boolean circuit. Although any boolean circuit can be represented in a restricted basis with only two boolean operators (e.g. AND and NOT is a complete basis) we provide more boolean operators in order to ease the building of integer operations. Objects of an instantiated `SlicedInteger` template are compatible with the basic integer used as template parameter, which allows the generation of a boolean circuit from common code. Except that only variable declarations must be changed from basic integer type to `SlicedInteger`. Some important compiler features must be respected in the implementation of the `SlicedInteger` class, such as signedness and integer conversions. Conversion operations from basic integer types to `SlicedInteger` type are provided. In this way it is possible to combine and integrate seamlessly basic integer types with `SlicedInteger` in a C++ algorithm, thus non-encrypted and encrypted variables (in agreement with FHE semantics).

When a sliced integer is instantiated from a constant, all its bits refer to the corresponding constant input vertex in the boolean circuit. The only way to define a variable value for the bits of a sliced integer is to read it from a standard C++ stream object. Doing so creates new input vertices in the boolean circuit and the read bits refer to these new vertices. Writing a bit to a standard C++ stream creates new

[1] Multiplicative depth is the number of sequential homomorphic multiplications which can be done on freshly encrypted ciphertexts in order to be able to decrypt and retrieve the result of multiplications.



```
#include <iostream>
#include <stdint.h>
#include "integer.h"
void f
  (std::istream &i,
   std::ostream &o)
{
  SlicedInteger<int8_t> a,b;
  i >> a >> b;
  o << a;
}
```

(a) Instantiation of input and output nodes in boolean circuit. The execution of the C++ code on the left creates the boolean circuit on the right. Inputs have a white border and outputs a black one. Edges are added to refer the written bits.

```
#include <iostream>
#include <stdint.h>
#include "integer.h"
void f
  (std::istream &i,
   std::ostream &o)
{
  SlicedInteger<int8_t> a,b;
  i >> a >> b;
  b = b ^ 0x01;
  a &= b;
  o << a;
}
```

(b) Instantiation of gate nodes in boolean circuit. The execution of the C++ code on the left creates the boolean circuit on the right. Input nodes corresponding to C++ input variables are treated using boolean XOR and AND gates.

Figure 1: Boolean circuits generation from C++ code examples.

output vertices in the boolean circuit and the written bits refer to these new vertices. The number of created (input or output) nodes correspond the the bit-size of the instantiated `SlicedInteger` class. Sub-figure 1a gives an example of this concept.

Performing operations on the sliced integers creates gate vertices in the boolean circuit. When a logic operation is applied to two bits, a gate vertex is created. The resulting bit refers to this new gate vertex, and the vertices referenced by the input bits are added as predecessors to the new gate vertex. Sub-figure 1b gives an example of this concept.

In our interface header file *integer.h*, we define an integer type per possible integer size. The definition of these types depends on a configuration flag *CONFIG_INTEGER*, which allows to switch from a normal algorithm to a sliced version one transparently: *EXECUTE* – the integer type is the same as the basic integer type, execution goes as usual and allows normal developping/debugging of the algorithm; *COMPILE* – the integer type is an instantiation of the `SlicedInteger` class with the corresponding basic integer type, the execution tracks the operations and builds the boolean circuit.

The implementation of algorithms using encrypted variables has an issue: the data-dependent control. Control flow statements (conditionals, loops) are accepted by our compiler only with non-encrypted data parameters. The following things are impossible: conditional instructions (jumps) with an encrypted integer in the condition and data dependent loops. These limitations are inevitable, otherwise some information on the plaintext of the variables would leak from the control flow. For example one could find encrypted variable value in a loop condition from the number of loop iterations.

It can seem that this issue seriously limits the usefulness of the compilation chain but as we shall see further several operations which depend on data are possible. For conditional assignment, as in the C++ ternary instruction `d=c?a:b`, the problem is solved by providing function $select(a, b, c)$ which returns a when c equals to one and b when c equals to zero. This function is implemented using boolean expression $(c\&a\,[i]) \oplus (\bar{c}\&b\,[i])$ applied on each bit i of variables a and b. Conditional assignment is equivalent to a two-input multiplexer circuit. Array dereferencing (with an encrypted index) is a generalization of conditional assignment where the condition variable has multiple values. Array dereferencing can be done by creating an array object which when dereferenced would insert several multi-input multiplexers. Equivalent circuits are described in [17]. Even under these limitations we were able to implement several real life applications. One example is the AES cipher which we will describe later.

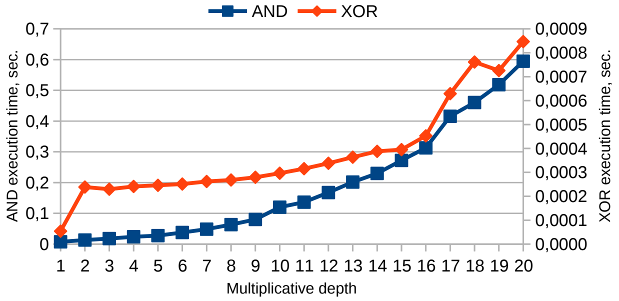


Figure 2: Execution times of homomorphic operations in function of multiplicative depth. Measurements have been performed using FHE scheme [10] (parameters: security $\lambda = 128$, cyclotomic polynomial $x^{1024} + 1$).

3.2 Middle end

Homomorphic encryption schemes must be parametrized in function of the multiplicative depth of circuits to evaluate. Execution time of a homomorphic multiplication (AND gate) is significantly larger than that of an addition (XOR gate). In figure 2 are represented empirical measurements of homomorphic operations execution times. Homomorphic addition is more than 100 times faster than homomorphic multiplication. More details about our implementation are given in section §3.3. In what follows we shall ignore the number of XOR gates in the circuit because their execution time is significantly smaller when compared to execution time of AND gates.

The middle-end phase of the compilation chain aims at optimizing the boolean circuit obtained from the C++ code generated in the precedent phase. The purpose is to decrease the total execution time of the boolean circuit on encrypted data.

The primary objective will be to minimize the multiplicative depth of the boolean circuit. Reducing the multiplicative depth of a circuit allows to decrease the execution time of every AND gate. It is possible to reduce the multiplicative depth at the price of an increased number of AND gates as long as the total execution time is smaller[2]. A secondary

[2] For example if the multiplicative depth of a circuit decreases from 8 to 7 then the number of AND gates could increase at most by 30% and the resulting execution time will not increase. We have used execution time measures from figure 2.

objective will be to minimize the number of logical AND gates in the boolean circuit. The multiplicative depth must not increase in the latter case.

In the literature one can find lots of works on boolean circuit optimization in the field of hardware synthesis (FPGA/ASIC). Circuit optimization algorithms from hardware synthesis have objectives and constraints that differ from the optimization needed in our case. For example XOR gates implemented in hardware are more expensive than the AND gates, which is completely in contradiction with optimization on boolean circuits for FHE. In hardware synthesis the propagation delay can be an optimization goal besides circuit size. The propagation delay is equivalent to the circuit depth computed with distinct propagation delays on each gate. The minimization of the circuit depth is not equivalent to the minimization of the circuit multiplicative depth.

Despite the fact that hardware boolean circuit optimization tools are not well suited for FHE boolean circuit optimization we have, as a first approach, adapted existing tools to our case. Besides, doing so allows us to take advantage from the existing expertise in the hardware synthesis domain. An open source software system used for hardware synthesis is ABC [2]. ABC software system comprises a set of tools used for synthesis and verification of boolean circuits. It is an open-source environment providing implementations of the state-of-the-art combinational and sequential synthesis algorithms.

ABC optimization tools are based on And-Inverter Graphs (AIGs). AIGs are logic circuits with two-input AND gates and inverters (negations) on edges. Logic circuit with only one gate type promise to decrease the complexity and the search space of circuit optimization algorithms. For more details about AIGs and ABC optimization tools refer to [20]. ABC reads different circuit formats and transforms them into AIG representation before applying various optimization algorithms. The front-end of our compilation chain exports boolean circuits in *blif* format [1]. The same format is used for boolean circuit outputted by the middle-end, once optimized.

We can assume that the AIG representation is beneficial to the optimization of FHE boolean circuits. ABC optimization tools have two objectives: circuit size and delay minimization. The size of an AIG circuit equals to the number of AND gates it contains and the delay is the length of the longest path from input to output nodes. At first sight it seems that this corresponds perfectly to optimization objectives for FHE boolean circuits, as AIG size correspond to FHE circuit size (number of AND gates) and AIG delay corresponds to FHE circuit multiplicative depth. Unfortunately this is not quite true. A counterexample is the XOR gate which corresponds to 3 AND gates and depth 2 circuit when transformed to AIG.

ABC package is script based. ABC takes a succession of optimization steps as input and applies them onto a boolean circuit. After several empirical studies we have deduced that the following optimization script gives satisfactory optimization results:

```
resyn2
if -g -K 15 -C 1024
resyn2
if -g -K 15 -C 1024
map
```

First step (`resyn2`) is a re-synthesis command which aims at minimizing circuit size as primary objective and circuit depth as secondary objective. Next command (`if`) aims to decrease circuit depth even if the circuit size increases lightly [21]. These operations are applied twice. Boolean circuit optimization for homomorphic execution allows a non-negligible increase in circuit size when the multiplicative depth of the circuit lowers. In ABC we were not able to express such type of constraints.

The last command `map` performs standard cell mapping of the AIG circuit. In this way we hope to additionally decrease circuit size and depth by mapping 3 AND gates to one XOR (inverse transformation described earlier). We use a cell library containing AND, OR, XOR and NOT gates. ABC was unable to perform cell mapping using a library with AND and XOR gates only. We remind that an OR gate can be expressed using one AND gate and several XORs, so an OR gate is equivalent to an AND gate for FHE circuit minimization objectives. A NOT gate can be expressed using a XOR. The gates AND, OR have unit size and delay and XOR, NOT gates have zero size and delay. The mapping algorithm will be forced to use less AND and OR gates in this case.

The execution time of ABC is not an issue for medium-sized circuits we have tested. Although one could sacrifice more time on optimization steps for gaining in homomorphic circuit execution time.

Circuit multiplicative depth and size in terms of AND gates is not always minimized by the previous ABC optimization script, sometimes it even increases. When building the AIG graph from the initial circuit each XOR gate is replaced by an equivalent sub-graph of only AND gates. The optimizations performed by the ABC tool minimize the total number of AND gates disregarding inherent structures representing XOR gates. The AND gates of these structures are potentially merged/simplified so that the initial XOR gates cannot be recovered by the mapping step. In this case we apply only simple redundancy removing optimizations (`balance`) so that XOR gates could be recovered during the mapping step. The following script is used:

```
balance
map
```

3.3 Back-end

The back-end takes as input an optimized boolean circuit from the middle-end step and generates a binary which executes the boolean circuit homomorphically. Additionally, in this step, are generated encryption, decryption and key-chain generation binaries. The generated binaries are linked to a homomorphic encryption library which is dimensioned dynamically to support the multiplicative depth of the boolean circuit to execute.

We have implemented the FHE scheme described in [10] without the bootstrapping step. The ciphertexts in this scheme are polynomial ring elements. Using the Chinese Remainder Theorem several plaintext bits can be packed into a ciphertext, see more details on this batching technique in [13]. The number of packed bits depends on the cyclotomic polynomial defining the ciphertext polynomial ring. In the actual configuration of our back-end only SIMD execution is supported (no ciphertext slot permutation has been implemented). The dimensioning of this FHE scheme is done automatically in function of the multiplicative depth, the desired security level (λ parameter) and eventually on the number of plaintext bits to pack into a ciphertext. Cyclotomic polynomial operations used in the dimensioning procedure (e.g. polynomial factoring used for batching) are performed using Sage [22]. The dimensioning parameters (cyclotomic polynomial defining the polynomial ring, ciphertext coefficient size, random distribution parameters, etc.) are read at execution from a configuration file. The *flint [15]* library is used to perform polynomial operations. Our FHE scheme has a generic interface. Other FHE schemes implementing this interface can be seamlessly incorporated into the compilation chain.

The boolean circuit generated by the middle-end has 4 types of gates: AND, OR, XOR and NOT. Each OR and NOT gate are replaced by an equivalent circuit composed of AND and XOR gates. The negation gate is replaced by a XOR gate on of whose inputs is one. The OR gate has two equivalent representations: (i) $a \parallel b = (a\&b) \oplus a \oplus b$ with 2 XORs or (ii) $a \parallel b = ((a \oplus 1) \,\&\, (b \oplus 1)) \oplus 1$ with 3 XOR gates. We use the one with 3 gates. Although one more XOR gate is used, these XORs are performed with a non-encrypted input (constant value one), which in FHE schemes are lighter.

A C++ code is generated from the boolean circuit composed of AND and XOR gates. Each gate is executed using a call to a respective function from the homomorphic encryption library. The generated C++ corresponds to either sequential circuit execution or parallel (for shared memory platforms) circuit execution. In the latter case circuit gates are scheduled off-line using a list schedule algorithm. We use a FIFO priority queue for dispatching gate executions. In the tests we have performed, the priority function did not significantly changed circuit execution time. The number of parallel execution threads to use is explicitly specified by user.

4. AES IMPLEMENTATION EXAMPLE

In this section we are going to show the easiness of implementing the AES-128 cipher using our compilation chain. We start by a brief overview of the AES cipher.

The AES is a block cipher which takes a 128-bit input data block, a 128-bit cipher key and outputs a 128-bit output. It consists of 10 equal rounds (with different round keys) applied sequentially on the input data block transformed into a 4×4 state matrix of bytes. Each round is divided into 4 steps which are applied on the state matrix: `AddKey`, `SubBytes`, `ShiftRows` and `MixColumns` (not applied at last round). In what follows we describe the steps used in encryption mode. `AddKey` is a XOR of the state matrix with the current round key. The `ShiftRows` step is a rotation to the left of state matrix row k by a $k-1$ places. `SubBytes` or S-box operation performs an inversion in the finite field $GF\left(2^8\right)$ followed by an affine transformation. Finally, the `MixColumns` step multiplies (finite field $GF\left(2^8\right)$ multiplication) the state matrix by a predefined 4×4 matrix.

Round keys are obtained from the cipher key using a *key expansion* or *key schedule* procedure. From the cipher key rk_0 (the cipher key acts as the first round key) 10 more 128-bit keys $rk_1,\ldots,rk_{10}$ are derived, one for each round. The cipher key rk_0 is used for the additional `AddKey` step applied on input data. Key expansion starts from the cipher key arranged in a 4×4 matrix. The first column of the next round key rk_p is obtained by adding together (additions are done in a finite field, i.e. bitwise XOR) the first and the modified last column of round key rk_{p-1}. The modification of the last column (`ScheduleCore`) consists in rotating the column one byte to the left. Applying S-box operation on each column element and adding to the first column element 2 exponentiated to the round number (in Rijndael's finite field). Other columns (2, 3 and 4) of rk_p are obtained by simply adding previous column from rk_p to the column on the same position from rk_{p-1}.

The decryption of an AES encrypted block is done equivalently by applying a set of inverse round steps in the opposite direction. The round keys are computed using the same key expansion procedure and are respectively applied in reversed order. Key expansion procedure does not change in the standard implementation of AES regardless of the used mode (encryption or decryption). The multiplicative depth of the AES-128 encryption is 40 together with the key expansion, whereas in decryption mode the multiplicative depth is already 80. This is due to the fact that the last round key rk_{10} (which is used first in decryption) has already a multiplicative depth of 40. In order to decrease the multiplicative depth in this mode we have implemented a different key expansion procedure for decryption, called further on decryption key expansion. Decryption key expansion starts from the last round key rk_{10}, which is read as input, and performs the usual key expansion in the opposite direction. The multiplicative depth of AES decryption with the modified key expansion procedure gets down to 40. This modification of the AES algorithm does not alter its security.

We have implemented the AES-128 in different execution modes: encryption or decryption with derived round keys (key schedule) or read round keys. The C++ implementation of AES is done at bit-level. Optimized circuits from [5] are used for S-box and reverse S-box[3]. The Armadillo compiler builds a boolean circuit from the C++ code. There are 128 input nodes in the boolean circuit for the input data block and either 128 input nodes for the initial round key (rk_0 for encryption/rk_{10} for decryption) or 1408 for the expanded round keys. The number of output nodes is 128 (one for each bit in the output data block). The generated boolean circuit has 5440 AND nodes for the encryption/decryption rounds and 1360 AND nodes for the key expansion part. As said earlier the multiplicative depth is 40 in all the modes.

Further on we provide a C++ code sample, used as input to our compiler, which performs the key schedule procedure in encryption mode (`KeySchedule` function):

```
void KeySchedule(Integer8** key, Integer8***
    roundKeys) {
  Integer8 temp[4];

  /* First round key is the cipher key itself */
  for (unsigned int c = 0; c < 4; c++) {
    for (unsigned int r = 0; r < 4; r++) {
```

[3] A straightforward S-box implementation using a truth-table has given a multiplicative depth of 10 after ABC optimizations. We had decided to use the optimized by hand S-box circuit because an AES circuit with a multiplicative depth 100 would have given an execution ˜10 times slower (estimation) than for the multiplicative depth 40.

```
      roundKeys[0][r][c] = key[r][c];
      if (c == 3) temp[r] = key[r][c];
    }
  }

  /* Compute next rounds keys from previous ones */
  for (unsigned int rnd = 1; rnd < 11; rnd++) {
    for (unsigned int c = 0; c < 4; c++) {
      if (c == 0) ScheduleCore(temp, rnd);
      for (unsigned int r = 0; r < 4; r++) {
        roundKeys[rnd][r][c] =
            roundKeys[rnd - 1][r][c] ^ temp[r];
        temp[r] = roundKeys[rnd][r][c];
      }
    }
  }
}
```

The C++ `KeySchedule` function takes as input a pointer `key` to the cipher key arranged in a 4×4 matrix. Parameter `roundKeys` is used to output derived round keys. `Integer8` is a typedef for a 8-bit `SlicedInteger` variable type. As we can see the implementation is simple and straightforward. If we replace `Integer8` by a standard 8-bit variable type (for example `unsigned char`) the same C++ code will compute round keys on non-encrypted data.

We have executed the AES-128 decryption algorithm with reverse key schedule (generated by the Armadillo back-end) on a mid-end 48-core server (4 x AMD Opteron 6172 processors with 64GB of RAM). No ciphertext batching is employed. Although batching will permit to substantially increase the throughput (i.e. the number of processed AES blocks per unit of time) the latency will increase too. Refer to [18] for a more detailed discussion of latency versus throughput in the case of homomorphic encryption. FHE scheme security parameter λ is 128 and the used cyclotomic polynomial is $\Phi_{2048}(x) = x^{1024} + 1$. Other FHE parameters are derived automatically following the procedure described in [10]. Obtained polynomial coefficient size is approximatively 23kbits, which corresponds to a 5.6MB ciphertext. The FHE key generation procedure takes 18 sec., FHE encryption of one bit 8 sec. and FHE decryption of one bit 0.2 sec. The obtained execution time for AES decryption is of approximatively 18 minutes, RAM memory usage is under 40GB. The same execution time is obtained for the AES encryption algorithm.

5. RELATED WORKS

Several domain specific languages for secure multi-party computation (SMC) have been proposed in the literature, an non-exhaustive list of such includes [7, 16, 4, 11, 3]. SMC is a cryptographic model in which n parties compute a common function, for example Yao's garbled circuits. In these models the communication between parties is proportional to circuit size to evaluate, whilst in FHE schemes the communication is proportional only to input data size. VIFF [7] is a framework built on-top of Python language which allows to easily specify SMCs. The arithmetic (boolean) circuit to execute is specified by the user. No circuit optimization is done by the framework, so it is up to the user to do this. The CBMC-GC [16, 11] is a C language compiler and framework for performing secure two-party computations (STC). It is an extension of the bit-precise model checker used to verify ANSI C source code. CBMC-GC transforms a C program into an optimized boolean circuit which can be executed by a STC platform (garbled circuits). Sharemind [4, 3] is a framework for MPC. It can be seen as a virtual machine which perform multi-party computations. Applications can be written in a high-level language SecreC (C extension) or in assembly language for the Sharemind virtual machine. The Armadillo front-end described in this paper most closely resembles to the VIFF framework. Both systems use a programmatic extension of an high-level language (C++ for Armadillo and Python for VIFF) to facilitate the use of cryptographic constructions (MPC and FHE). Armadillo's front-end also automatically bit-slices high-level C++ instructions. Unlike the VIFF framework, in Armadillo a middle-end is used to optimize obtained boolean circuits.

Homomorphic execution implementations for the bit-sliced AES decryption algorithm were previously reported in [14, 8]. The authors implemented the AES decryption but no key schedule procedure is done. The 11 round keys are inputs to the AES circuit. The size of the boolean circuit is smaller by 25% in this case. Without the key schedule procedure the authors did not have to cope with the multiplicative depth 80 of the usual AES decryption implementation. A direct comparison of execution performance between these implementations and our is inappropriate because of different FHE schemes. One of the objectives of [14, 8] was to increase the AES execution throughput by using batching techniques. In contrast to this, our objective was to decrease the latency. That is why we have employed explicit parallelization of boolean circuit execution and no batching. We shall also note that both these implementations are using the NTL library for polynomial arithmetic. The NTL library has a restricted support for multi-threading, so an explicit parallelization of boolean circuit execution is most likely impossible in current conditions. And last but not least, we have implemented the AES algorithm in a high-level language, compared to the manual AES circuit implementation.

6. CONCLUSIONS AND PERSPECTIVES

In this paper we have presented the Armadillo compilation chain used for compiling privacy-preserving applications. The compilation chain consists of 3 phases: high-level language (C++) code transformation to a boolean circuit (front-end), optimization of the boolean circuit (middle-end) and execution of this circuit on encrypted data using homomorphic encryption (back-end). The implementation of applications using the homomorphic encryption in back-end becomes easier with Armadillo. We have implemented the AES-128 algorithm in order to show this. The execution latency of AES algorithm is only 18 minutes which represents an advancement compared to existing homomorphic AES implementations. For the AES decryption algorithm we have introduced a modification in the key schedule procedure in order to decrease the multiplicative depth of circuit from 80 to 40. Although the compilation chain allows to build homomorphic encryption based applications and seems complete much future work have to be done. In what follows we shall elaborate some perspectives which seem promising to us.

We have restricted ourselves to binary operations (boolean AND and XOR) of FHE primitives, although FHE allows to perform operations homomorphically on integers modulo t for $t > 2$ or more generally in other finite fields. This aspect of homomorphic encryption is not taken into account in our compilation chain. Using for example $\mathbb{Z}_{256}$ as plaintext permits to execute homomorphically addition and multiplication operations modulo 256 directly, which for some applications will provide a performance increase. Another aspect of our front-end is that the high-level code is directly transformed into a boolean circuit, although passing by some sort of intermediate representation (e.g. arithmetic circuit) will provide more optimization possibilities and potentially a smaller boolean circuit afterwards.

The current middle-end uses existing boolean circuit optimization tool (ABC) from the field of hardware synthesis. As said earlier the objectives of circuit optimization for homomorphic encryption and hardware synthesis differ. That is why we execute two optimization scripts and keep the obtained circuit which has the smallest multiplicative depth. We think that there is more research to be done in this direction, thus on optimization of boolean circuits with multiplicative depth as primary objective and number of multiplications as secondary objective.

Available FHE libraries in the compiler back-end should be diversified in order to be able to chose the FHE library which is the most adapted to the developed application. The current parallel boolean circuit execution back-end supports only shared-memory architectures. A promising perspective will be the development of a back-end for distributed-memory architectures.

7. REFERENCES

[1] Berkeley Logic Interchange Format (BLIF). University of California, Berkeley, July 1992.

[2] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification, Release 30308. http://www.eecs.berkeley.edu/˜alanmi/abc/.

[3] D. Bogdanov, R. Jagomägis, and S. Laur. A Universal Toolkit for Cryptographically Secure Privacy-preserving Data Mining. In *Proceedings of the 2012 Pacific Asia Conference on Intelligence and Security Informatics*, PAISI'12, pages 112–126, 2012.

[4] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 192–206, 2008.

[5] J. Boyar and R. Peralta. A Small Depth-16 Circuit for the AES S-Box. In *SEC*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 287–298, 2012.

[6] Z. Brakerski, C. Gentry, and S. Halevi. Packed Ciphertexts in LWE-Based Homomorphic Encryption. In *Public Key Cryptography*, pages 1–13, 2013.

[7] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography: PKC '09*, pages 160–179, 2009.

[8] Y. Doroz, Y. Hu, and B Sunar. Homomorphic AES Evaluation using NTRU. Cryptology ePrint Archive, Report 2014/039, 2014. `http://eprint.iacr.org/`.

[9] P. Dubrulle, S. Carpov, and R. Sirdey. Automatic Bitslicing of Algorithms for VLIW Architectures. In *The 51st Design Automation Conference (DAC), WiP session*, 2014.

[10] J. Fan and F. Vercauteren. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[11] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. volume 8409 of *Lecture Notes in Computer Science*, pages 244–249.

[12] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, pages 169–178, 2009.

[13] C. Gentry, S. Halevi, and N.P. Smart. Fully Homomorphic Encryption with Polylog Overhead. In *EUROCRYPT*, pages 465–482, 2012.

[14] C. Gentry, S. Halevi, and N.P. Smart. Homomorphic Evaluation of the AES Circuit. In *CRYPTO*, pages 850–867, 2012.

[15] W. Hart, F. Johansson, and S. Pancratz. FLINT: Fast Library for Number Theory, 2013. Version 2.4.0, `http://flintlib.org`.

[16] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure Two-party Computations in ANSI C. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 772–783, 2012.

[17] M. Kwan. Reducing the Gate Count of Bitslice DES. *IACR Cryptology ePrint Archive*, 2000:51, 2000.

[18] T. Lepoint and M. Naehrig. A Comparison of the Homomorphic Encryption Schemes FV and YASHE. In *AFRICACRYPT 2014*, volume 8469 of *Lecture Notes in Computer Science*, pages 318–335, Marrakesh, Morocco, 2014.

[19] C. A. Melchor, S. Fau, C. Fontaine, G. Gogniat, and R. Sirdey. Recent Advances in Homomorphic Encryption: A Possible Future for Signal Processing in the Encrypted Domain. *IEEE Signal Process. Mag.*, 30(2):108–117, 2013.

[20] A. Mishchenko, S. Chatterjee, and R. K. Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *DAC*, pages 532–535. ACM, 2006.

[21] A. Mishchenko, S. Cho, S. Chatterjee, and R. K. Brayton. Combinational and sequential mapping with priority cuts. In *ICCAD*, pages 354–361, 2007.

[22] W. A. Stein et al. *Sage Mathematics Software (Version 6.4.1)*. The Sage Development Team, 2014. `http://www.sagemath.org`.