Parallel Implementation of GC-Based MPC Protocols in the Semi-Honest Setting

Mauro Barni[1], Massimo Bernaschi[2], Riccardo Lazzeretti[1], Tommaso Pignata[1], and Alessandro Sabellico[2]

[1] Information Engineering and Mathematical Science Department, University of Siena, Italy
barni@dii.unisi.it, {riccardo.lazzeretti,pignata.tommaso}@gmail.com,
[2] Institute of Applied Computing, National Research Council of Italy, Rome, Italy
{massimo.bernaschi, a.sabellico}@gmail.com

Abstract. Parallel computing offers the chance of improving the efficiency of Garbled Circuit technique in multi-party computation protocols. We propose two different types of parallelization: fine-grained, based on the parallel evaluation of gates, and coarse grained, based on the parallelization of macro-blocks. To analyze the efficiency of parallel implementation, a biometric scenario, having an intrinsically parallel nature, is considered. Moreover our approach is compared to previous works by using a privacy preserving implementation of AES encryption. We show that both fine-grained and coarse-grained solutions provide significant runtime improvements. Better results are obtained by the coarse-grained parallelization, which, however, can be exploited only when the same block is used more than once in parallel, whereas fine-grained parallelization can be applied to any garbled circuit.

Keywords: Parallel Computing, Multi-Party Computation, Signal Processing in the Encrypted Domain, Garbled Circuits

1 Introduction

Rapid technological advances in multi-party signal processing have given rise to a variety of new signal processing applications for which security aspects can no longer be dealt with by classical cryptographic methods. The classical security model is targeted toward protecting the communication of two trusted parties against a potentially malevolent third party. In such cases, it is sufficient that secure cryptographic primitives are applied on top of transmission and processing modules. In an increasing number of applications, however, the classical security model is no longer adequate since at least one of the parties involved in the communication, distribution or processing of the data may not be trustworthy.

Multi-Party Computation (MPC) provides a clever way to process data without revealing any details about the data itself during the processing. When the to-be-processed data is a signal [11,20], MPC is customarily referred to as S.P.E.D. (Signal Processing in the Encrypted Domain), since signal protection is usually achieved by encrypting the signals and processing them in encrypted form. Possible applications of MPC are virtually endless. For example, a database server may be untrustworthy [1], creating the need to hide the content of queries to the database server, while still allowing the query to be resolved. As another example, we may consider a remote diagnosis service [4,21], where a non-trusted party is tasked with processing sensitive medical data without leaking the private data of the patient (including the diagnosis results). The use of MPC for biometric identification and access control is also gaining popularity since it permits to protect the privacy of the biometric owners in client server applications. Many types of biometries have been addressed in S.P.E.D. analysis, including: face recognition [10], fingerprinting identification [2,3], iris identification [23], etc. Besides the scenarios already outlined, we also recall user preferences [12], watermarking [9], digital rights management [14].

In a two-party computation (2PC) protocol, two players, usually referred to as Alice and Bob, are interested in cooperating to evaluate a given public function $z = f(x;y)$, where x and y are the inputs owned by Alice and Bob respectively, and where neither Alice nor Bob wants to disclose her/his inputs to the other party. At the end of the protocol, the output will be available to one party between Alice and Bob, or to both of them. Yao's Garbled Circuits theory (GC) [30,31] is one of the most used approaches to private computing. In its seminal work, Yao showed that any polynomial size functionality $f(\cdot)$ can be evaluated privately in a constant number of rounds, with polynomial communication and computational overhead. GC allows the evaluation of the binary circuit implementing $f(x,y)$ on input bits privately owned by Alice and Bob, so that the final result is available to one of them (or both), whereas intermediate values cannot be discovered by any of the parties. Yao's protocol has long been thought to be of theoretical interest only due to its complexity. However, recent works have shown several ways to improve GCs efficiency, making them usable even in practical scenarios. Parallel evaluation of circuits is surely one of such methods, however even though it is known that GC can benefit from parallelization, no benchmark analysis has been provided before.

In this paper we describe two ways whereby parallel computing can significantly improve the GC efficiency. Parallel computing has been used in scientific applications for decades in fields like fluid dynamics, material science, weather forecasts. Recently, due to the difficulties of further reducing the clock rate of the processors, all CPU vendors are investing, for the sake of better performance, on multi and many-core architectures, so parallel processing is becoming common practice in many other fields. Due to specific features of GC, briefly recalled in Section 2, parallel processing of GC requires a paradigm that entails an overhead as limited as possible for the management of parallel tasks.

Related Work. Several implementations of GC have been already proposed, starting from Fairplay [24], FairplaySPF [27], Tasty [15], etc. To the best of our knowledge, currently, the most efficient implementation in the semi-honest setting is the one presented in [16].

Recently also some parallel implementations have been proposed. In [29], GPU is used for parallel implementation of specific operations needed by the GC protocol, whereas [13] uses GPU processors for GC implementation in the malicious setting.

Our Contribution. In this paper we demonstrate that GC can take advantage of parallel computation, especially when the representation of the required functionality results in a very large circuit. We address two different types of parallelization: the first one, fine-grained parallelization, is based on the parallelization of the single gates composing a circuit, while the second one, coarse-grained parallelization, is based on macro-blocks parallelization. The proposed solutions are evaluated by running them on multi-core processors. In particular, we resort to *threads* for parallel processing of GC since they run very efficiently on modern CPUs and offer all the synchronization mechanisms required to prevent race conditions in the evaluation of GC. For our tests a biometric identification scenario has been chosen, for its high parallel nature. Moreover, to compare our results to previous implementations, a privacy preserving implementation of AES encryption [8] has been tested as well.

Outline. In Section 2 the basis of the GC scheme is presented; in Section 3 we present our parallel implementations, whose application to privacy preserving biometric scenarios and AES encryption is presented in Section 4, together with the obtained results and a security analysis. Finally some conclusions are provided in Section 5.

2 Preliminaries

Garbled circuit (GC) is an elegant method for secure function evaluation of boolean circuits. The general idea of GCs, going back to Yao [30,31], is to encrypt (*garble*) each wire and gate with a symmetric encryption scheme.

Yao's Protocol At a high-level, Yao's GC protocol works as follows: in the setup phase, the *constructor* (Bob) generates an encrypted version of the function f (represented as boolean circuit), called *garbled circuit* $\widetilde{f}$. To that purpose, he assigns to each wire w_i of f two randomly chosen garbled values $\widetilde{w}_i^0, \widetilde{w}_i^1$ (symmetric keys) of t bits each (security parameter set equal to $t = 80$ for short-term security), that correspond to the respective values 0 and 1. Note that $\widetilde{w}_i^v$ does not reveal any information about the plain value v as both keys look random. Then, for each gate of f, the constructor creates helper information in form of a *garbled table* $\widetilde{T}_i$ that allows to decrypt only the output key from the gate's input keys. Each table is used to find the correct value of the output wire of the gate given a specific value on each of the garbled gate's input wires. By expressing the functionality of a given gate as $\gamma = G(\alpha, \beta)$, where $\alpha \in \{0,1\}$ and $\beta \in \{0,1\}$ are the input wires of the gate while $\gamma \in \{0,1\}$ is the gate's output wire, then the garbled computation table is a random permutation of $E_{\widetilde{w}^\alpha}\left(E_{\widetilde{w}^\beta}(\widetilde{w}^\gamma|check)\right)$ for all the four possible input pairs, (α, β), using some symmetric encryption function $E_{key}(\cdot)$ and appending a *check* sequence to the garbled output that helps the identification of the correct row.

The garbled circuit $\widetilde{f}$, consisting of the garbled tables generated from the gates, is sent to the *evaluator* (Alice). Later, in the online phase, Alice obliviously obtains the garbled values $\widetilde{x}$ and $\widetilde{y}$ corresponding to the plain inputs x and y of Alice and Bob, respectively. To convert a plain input bit y_i of Bob into its garbled version, Bob simply sends the key $\widetilde{y}_i^{y_i}$ to Alice. Similarly, Alice must obtain the garbled secret $\widetilde{x}_i^{x_i}$ corresponding to her input bit x_i, avoiding that Bob learns x_i. This can be achieved by running, possibly in parallel, for each bit x_i of x, a 1-out-of-2 *Oblivious Transfer (OT)* protocol [25]. OT is a cryptographic protocol taking as input Alice's choice bit $b = x_i$ and Bob's strings $s^0 = \widetilde{x}_i^0$ and $s^1 = \widetilde{x}_i^1$. The protocol guarantees that Alice obtains only the chosen string $s^b = \widetilde{x}_i^{x_i}$ while Bob learns no information on $b = x_i$. Afterwords, Alice evaluates the garbled circuit $\widetilde{f}$ on $\widetilde{x}, \widetilde{y}$ by evaluating the garbled gates one-by-one decrypting the rows of the associated tables, where the correct decryption is identified by the *check* sequence. Finally, Alice obtains the corresponding garbled output values $\widetilde{z}$ which can be decrypted into the corresponding plain output $z = f(x,y)$.

OT implementation To efficiently implement OT, the following techniques are used:

Pre-computing OT [6] allows moving computation and communication burden to the setup phase, where both parties run the OT protocol on random inputs. This makes secrets generation independent from circuit execution. Then, in the more time-critical online phase, Alice and Bob use those random inputs to mask their real inputs with a one-time pad. OT secrets, that have been produced in the offline phase, are "consumed" by retrieving them from the files, where they have been stored by the offline generator procedure. The same secret is never used twice in the same or other circuits.

Extending OT efficiently [17] allows for the reduction of the computation complexity during the setup phase by replacing n parallel OTs of t-bit-strings with t parallel OTs of t-bit strings performed in the opposite direction, followed by other computations that extends the number of OT.

Implementation over elliptic curves permits the implementation of efficient OT protocols, evaluating n parallel OTs of ℓ-bit strings, implemented efficiently with the protocol of [25] over *elliptic curves*. The use of elliptic curves allows to perform operations on and transmit shorter cyphertexts with respect to group Z_n. Unfortunately the computation complexity of the protocol increases, but the communication complexity reduction results in a significant decrease of the execution time, since communication between parties is a critical component of the execution.

Optimized GCs While Yao's GC formulation does not take into account the problem of the efficiency, many improvements have been proposed in the last years. The principal improvements can be summarized as follows.

First of all for efficient implementation of GC, a random oracle $H(\cdot)$ is used. It is usually instantiated with a suitably chosen cryptographic hash function such as SHA-256 [26]. Hence symmetric encryption of the gate rows is performed as

$$E_{\widetilde{w}^\alpha}\left(E_{\widetilde{w}^\beta}(\widetilde{w}^\gamma|check)\right) = (\widetilde{w}^\gamma|check) \oplus H(\widetilde{w}^\alpha|\widetilde{w}^\beta|s) \tag{1}$$

where s is a gate identifier.

The *point and permute* technique [24] allows the evaluator to decrypt directly the correct row, resulting in a double advantage: only a single call to the encryption function for each gate is needed during evaluation and the *check* sequence is no longer necessary, reducing the dimension of the garbled tables. The idea is to associate a single permutation bit $\pi_i \in \{0,1\}$ to each wire i. The garbled value associated to the wire is $\widetilde{w}^i|c_i$, where $c_i = b_i \oplus \pi_i$. Each row of the garbled table is hence computed as $(\widetilde{w}^\gamma|c^\gamma) \oplus H(\widetilde{w}^\alpha|\widetilde{w}^\beta|s)$ and the rows of the garbled tables are permuted according to the input permutation bits. In such a way, during evaluation, the correct row is directly selected by observing c^α and c^β.

Another important improvement is the *free-XOR* technique [19]. Garbled XOR gates require no garbled table and negligible computation. A global key Δ is randomly chosen and the secrets for each wire i are generated so that $\widetilde{w}_1^i = \widetilde{w}_0^i \oplus \Delta$. The output wire of a XOR gate having input wires α and β is computed as $(\widetilde{w}^\gamma | c^\gamma) = (\widetilde{w}^\alpha | c^\alpha) \oplus (\widetilde{w}^\beta | c^\beta)$.

Finally Garbled Row Reduction [28] can also be used to reduce the size of non-XOR gates by eliminating a row in each garbled table, resulting in a $\approx 25\%$ reduction of non-XOR gate garbling, transmission and evaluation times.

Our implementation To evaluate the benefits provided by parallel evaluation, we implemented our version of GC tools. Our C++ implementation of Garbled Circuits relies on the object-oriented paradigm to guarantee reusable code and consistent modules interaction. An efficient implementation has been obtained by implementing the principal tricks presented above, except OT implementation over elliptic curves and Garbled Row Reduction. Even if such techniques would improve the protocol efficiency, their absence does not compromise the comparison between the sequential GC implementation and the parallel implementations.

We consider that by using the extending OT technique, blocks of 1 million OTs are precomputed and stored, so that when a given number of OT are evaluated online, the same number of precomputed OTs are picked, used and removed from the memory. In our implementation we evaluate ≈ 200000 offline OTs in a second. It is important to underline that we consider the function f that Alice and Bob are going to jointly evaluate, to be known before they have the input values, hence garbling and garbled circuit transmission can also be performed in the setup phase.

3 Circuit Parallelization

As mentioned above, many recent works have improved the efficiency of GC. Hereafter, we demonstrate that the evaluation of GC can also take advantage from parallel execution. Parallel processing can be used for both OT, where bits are independent from each other, and processing of those gates that, depending on the circuit, can be garbled/evaluated in any order.

With respect to other GC implementations, in OT parallelization, we have parallelized secrets generation, by computing multiple bits at the same time and the protocol used to securely exchange secrets, the Bellare-Micali protocol [7], whose computation is divided in offline and online phases. Gate parallelization strongly depends on the characteristics of the to be evaluated function, so a flexible and low-overhead parallelization technique is required. *Threads* fulfill both requirements: on multi-processor or multi-core systems, they can concurrently be assigned to each processor or core running a thread of the same process (or task) and the time required to create and synchronize them is much lower with respect to standard processes. *Threads* are supported at both language (*e.g.,* Java) and operating system level (*pthreads* in Unix-like OS and *winthreads* in Windows). In the present work we resort to *pthreads* programmed in C++ and the resulting GC evaluation engine runs seamlessly under Unix and Mac OS. Porting to Windows is possible simply by replacing the calls to *pthreads* with invocation of *winthreads* primitives.

Two different kinds of parallelization are considered: fine-grained, corresponding to classic parallelization of single gates evaluation and our new subdivision of the circuit into layers, where with layer we intend a subset of the circuit's gates that can be evaluated independently from other gates by Garbler and Evaluator; and coarse-grained, considered here for the first time where macro blocks composing the circuit are parallelized. Inside each macro block, gates can again be evaluated in parallel.

3.1 Fine-grained parallelization

In fine-grained parallelization, the gates of the circuit are subdivided into layers, such that all the gates in the same layer can be evaluated in parallel. No special attention is needed during circuit design, that is performed as usual. Later, the circuit is parsed, so that the gates are sorted to ease the parallel execution. The gates connected only to input wires are placed in layer 0. Then the gates having, at least, one input wire coming from a gate in layer 0 are placed in layer 1. The procedure is then iterated on all the gates, placing a gate having input wires obtained as output from two gates respectively already in layers i and j, in layer $\max(i, j)+1$. In the end of the scanning procedure, all the gates of the circuit are grouped in layers. Almost contemporaneously to us, a similar scheme for gate parallelization has been proposed also in [13], although their target platform are the Graphics Processing Units (GPU).

It is important to underline that the sorted circuit can be garbled and evaluated sequentially or by using threads that permit the parallel elaboration of gates in the same layer. This permits a sequential execution from a single core system, while in multi core systems, to prevent from incurring in a slow down caused by an insufficient work load for each thread, there is a minimum number of gates per thread that can be executed in parallel. If the number is lower than the threshold the execution is serialized. If we indicate with Δt the time that can be saved by running the level in parallel, we have the condition $\Delta t = S_t - (P_t + C) > 0$ if $S_t - P_t > C$ where P_t is the execution time in parallel, S_t is the serial execution time and C is the overhead introduced by the management of the threads (creation, synchronization, *etc.*).

Having different garbling/evaluation procedures, we analyze separately the parallelization of XOR gates and non-XOR gates. Thanks to [19], circuits are designed to reduce the number of non-XOR gates and, as a result, XOR gates are usually the most common gate type (*e.g.,* in our circuits 74%, on average). As expected, the computational burden necessary to execute them is less than that required by other gates. Nevertheless for large circuits there is such a high number of XOR gates per level to justify parallelization on this phase. As a matter of fact, we obtain a good speedup when executing in parallel XOR gates for large circuits. Non-XOR gates are generic gates that can have an arbitrary number of inputs and any truth table (usually plain gates are often used to execute AND, OR operations with 2 inputs). That class of gates has a major impact on computation time since, for each gate, it is necessary to cipher its truth table associated to the possible secrets' combinations on inputs. The ciphering requires the execution of a SHA-256 hash function and several XOR operations on the gate secret inputs. For non-XOR gates we have parallelized only the creation and the ciphering of the truth table. This is in charge of the Garbler that afterward sends the result through the communication channel. NOT gates, that can be also evaluated for free, are not very expensive in computational terms and also relatively few even for large circuits. As a consequence, we did not develop a parallel procedure for the execution of NOT gates.

In our solution we resort to CPU *threads* for parallelization since the grain of the computation hardly justifies the usage of hundreds of relatively slow cores like those available in a GPU. CPU *threads* have a very low creation overhead and can be managed in a dynamic way depending on the features of the circuit under evaluation (*i.e.,* the number of gates in each layer).

3.2 Coarse-grained parallelization

A macro block parallelization presents many advantages, but requires some substantial changes in the protocol. The possibility of dividing a circuit in blocks makes the circuit representation easier, since the developer can design and test small parts of the circuit. Moreover, there is no need to repeat the design of identical parts of the circuit, as often happens in protocols where the same operation is repeated on different inputs. Finally this solution reduces the overhead introduced by thread management, since, while in fine-grained parallelization in each layer a thread is created and then destroyed for each gate, here a thread is created for groups of gates.

To design a circuit by using macro blocks it is necessary to define also secret inputs and outputs, besides the classical evaluator and garbler inputs and outputs. A secret input is a sequence of bits, obtained as output from another macro block, that can not be revealed to either the garbler and the evaluator. In practice, secret inputs/outputs are used to connect different blocks. Obviously, in the design phase, particular attention must be paid to the dimension of secret inputs and outputs to avoid inconsistency problems. For a good design, it can be useful to handle the evaluator and garbler input association phase by using one or more blocks that accept plain inputs and return the associated secrets.

During garbling it is important to use the same global key Δ for all the circuits, then garbling is performed as usual, paying attention to the pair of input secrets. Obviously, if the same macro-block is used more than once in the circuit (with different secret inputs), each instance needs to be garbled independently from the others, because, as usual, if the same garbled circuit is evaluated twice with different inputs, the security of the protocol is compromised. Garbling of macro blocks that can be processed in parallel is assigned to different threads.

Evaluation is performed as usual, the only change consists in the requirement of storing the secrets obtained as outputs of a block to assign them to the inputs of another block. Macro blocks that can be evaluated in parallel are assigned to different threads.

Even if blocks evaluated in parallel can be different, when the same block is garbled/evaluated multiple times in parallel, the operations performed by the threads can be driven together, because they perform the same operation on different values. Beyond the easy design and the parallelization, this solution results in another, non negligible, advantage: the file containing the description of a macro block that is garbled/evaluated multiple times in parallel is read only once, reducing the memory load.

4 Analysis

To provide an analysis of the benefits introduced by the parallel evaluation we consider a biometric matching problem and AES encryption of a large amount of data. For both scenarios, we compare the sequential implementation to the parallel implementations (fine-grained, coarse grained and coarse-grained with fine-grained parallelization inside the blocks). We show how the results change as a function of the number of available threads and the time needed by each phase of the computation. Finally we provide a security analysis of the two implementations.

4.1 Iris identification

As first example, we consider the iris identification protocol proposed in [23], modified so that the final result is the index of the best match, if exceeding a given threshold, instead of a simple answer that specifies if the tested biometry is in the database or not. The parameters are chosen according to the original paper and their values are specified during the description.

In such protocol, the biometric server, Bob, has an iris gallery which stores the iris features $\{X_1,\dots,X_n\}$ of $n = 1023$ members. X_i is a binary vector denoted as $(x_{i,1},\dots,x_{i,\ell})$, where $\ell = 2048$. The user, Alice, provides a probe $q = (q_1,\dots,q_\ell)$ and evaluates the GC which produces a match if there exists at least an $i \in \{1,\dots,n\}$ such that $d(q,X_i) < \varepsilon$ for a similarity threshold ε. $d(q,X_i)$ is a modified Hamming Distance (HD) defined below:

$$d(q,X_i) := \frac{D(q,X_i)}{M(q,X_i)} = \frac{||(q \otimes X_i) \cap mask_q \cap mask_{X_i}||}{||mask_q \cap mask_{X_i}||}, \tag{2}$$

where $\otimes$ denotes XOR, $\cap$ AND, and $||\cdot||$ the norm of the binary vector; $mask_q$ and $mask_{X_i}$ are the corresponding binary masks that zero out the unusable portion of the irises due to occlusion by eyelids and eyelash, specular reflections, boundary artifacts of lenses, or poor signal-to-noise ratio. Considering that masks do not disclose sensitive information about the subjects, as demonstrated in [23], a common mask can be used. Mask filtering is performed in the plain domain on all the irises by Alice and Bob and together they can compute the distances

$$d'(q,X_i) := \frac{\mathrm{HD}(mask(q), mask(X_i))}{||CM||}, \tag{3}$$

where $\mathrm{HD}(\cdot)$ denotes the Hamming distance and $mask(\cdot)$ is the masking function with the common mask, identified by CM.

At this point, given an acceptance threshold ε, the index of the best match can be obtained as $\arg\min(\varepsilon, \{d'(q,X_i)\}_{i=1}^n)$. If the return value is equal to 0 there is no match. We underline that, for simplicity, we can reformulate the problem as

$$\arg\min(||CM||\varepsilon\ ,\ \{\mathrm{HD}(mask(q), mask(X_i)\}_{i=1}^n). \tag{4}$$

The protocol can be implemented by the circuit shown in Figure 1, where the Hamming distance is computed by XOR gates between the two inputs and a COUNTER circuit [5], whereas the argMIN tree is implemented as in [18]. The circuit is composed by approximately 6.3 millions of gates,1.1 millions of which are non-free gates.

While fine-grained parallelization is applied on a single circuit implementing Figure 1, segmented in 356 layers, coarse-grained parallelization needs subdvision into

sub-blocks. We can identify the following blocks, whose composition is shown in Figure 2:
− n *Garbler input interfaces* for Bob's iris templates, each one converting one ℓ-bit long input in ℓ t-bit long secrets;
− 1 *Evaluator input interface* for Alice's iris template query, converting one ℓ-bit long input in ℓ secrets;
- 1 *Garbler input interface* for acceptance threshold ε, converting one $\lceil \log_2 \ell \rceil$-bit long input in $\lceil \log_2 \ell \rceil$ secrets;
− n *Hamming distances*, each one having 2 inputs composed by ℓ secrets and 1 output composed by $\lceil \log_2 \ell \rceil$ secrets;
- 1 *argMIN tree*, having n inputs represented with $\lceil \log_2 \ell \rceil$ secrets and returning an index represented with $\lceil \log_2 n+1 \rceil$ bits (output interface is included in the block).

The index of the best match is obtained by a reverse argMIN tree having $\lceil \log_2 n+1 \rceil$ levels and $n+1$ inputs, where the input$_0$ is the threshold and the input$_i$ is the output of the i-th Hamming distance. The i-th level ($i = 0 \ldots \lceil \log_2 n+1 \rceil - 1$) is composed by $\lceil \frac{n+1}{2^{i+1}} \rceil$ MIN blocks. Each MIN selector circuit in level 0 outputs the secret relative to the minimum value together with a secret related to a bit signaling whether the minimum value is in the left (1) or in the right (0) input. The other MIN selector circuit in the tree has two input values coming from the two sub-trees connected to their inputs. These values are composed by the highest sub-tree's input and the relative position in the sub-tree. The MIN selector circuit outputs the secrets corresponding to the highest value concatenated with the subtree index, preceded by a bit assuming the value 0 whether the output comes from right subtree or 1 if it comes from left subtree. In such a way, we obtain the relative index of the new sub-tree. The MIN circuit in the final level outputs the plain index to Alice, Bob or both. Considering that the input bitlength changes at

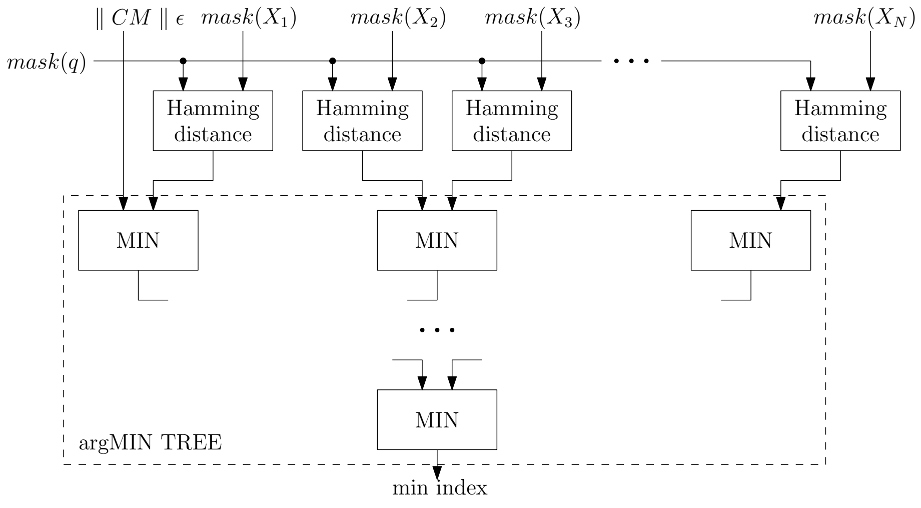

Fig. 1: Iris identification scheme.

each level, a different circuit has to be described for each level. It is important to note that if in a level the number of k inputs is odd, we need $k/2$ MIN selectors, but if k is even, $(k-1)/2$ max selectors are needed. In the second case the last value needs to be propagated to the next level and the bit 0 has to be concatenated. This can be done through a *false* MIN selector circuit, as shown in Figure 3. Hence in a level i each MIN block has 2 inputs represented by $\lceil \log_2 \ell \rceil + i$ secrets and 1 output represented by $\lceil \log_2 \ell \rceil + i + 1$ secrets. The final MIN block (level $\lceil \log_2 n + 1 \rceil - 1$) differs from the others because its output is composed by $\lceil \log_2 n + 1 \rceil$ bits.

It is important to note that all the sub-circuits placed in the same level in Figure 2 (input interfaces, HDs, MINs of level i) can be evaluated in parallel. Moreover only a description file for each block type is necessary.

We suppose that server and client perform garbling, OT precomputation and transmission of the garbled circuit offline, to provide the most efficient computation when real data is available. As already mentioned, OT precomputation is performed by using the OT extension protocol that, in our case, performs $\approx 1.000.000$ OTs offline in abount 5 seconds, hence the OT precomputation runtime reported in the table is the portion of the time referred to 2048 OTs (the same implementation is used for all the solutions). Tests have been performed on a system with two Intel Xeon E5-2609@2.4 GHz with 10 Mbytes of cache and 16 Gbyte of RAM connected to a Fast Ethernet network (100 Mb/sec.). Each ES-2609 has four cores, hence the total number of available cores is eight.

Table 1 shows the different implementation runtimes needed for each element of the protocol when 8 threads are used (except for sequential implementation). During the of-

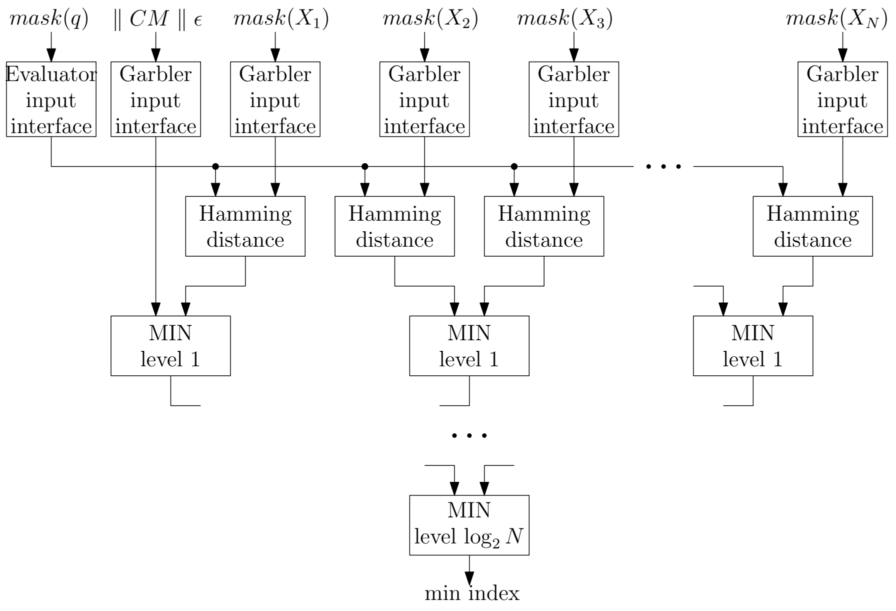

Fig. 2: Block subdivision of iris identification scheme for coarse-grained parallelization.

	Phase	S	FG	CG	CG+FG
Offline	Garbling	9.772	3.475	2.175	1.860
	OT Prec.	0.010	0.010	0.010	0.010
	Garbled table Tx	1.701	1.314	0.036	0.690
Online	Bob's secret Tx	0.338	0.378	0.130	0.158
	Alice's secret Tx	0.002	0.003	0.002	0.002
	Evaluation	3.437	2.899	1.019	1.765

Table 1: Runtimes (in seconds) of iris identification protocol by using sequential implementation (S), fine-grained (FG) parallelization, coarse-grained parallelization (CG), or both. 8 cores have been used in parallel implementations.

fline phase the same OT precomputation protocol has been used for all the solutions. We can easily observe that all the parallel solutions provide better runtimes with respect to the sequential solution. As expected, the parallelization of the single gates introduces a management overhead greater than the one introduced by macroblock parallelization. On the other hand, the use of gate parallelization inside parallelized macroblocks produces worst results with respect to the coarse-grained parallelization, but the solution is still preferable than fine grained parallelization.

Figure 4 shows the offline, online and total runtimes of the different implementations as a function of the number of threads. We can see that the performance increase with the number of threads, especially in the solutions that rely on coarse grained parallelization having a trend that is inversely proportional to the number of threads used. Indeed the results are affected by the number of cores available and their turnover due to the system inactivity time. We can observe that, having 8 cores, there is no more improvement if more than 16 threads are used.

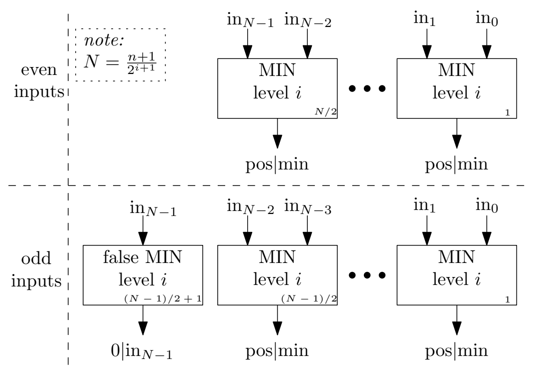

Fig. 3: Subtree level composition.

4.2 AES encryption

As a second test case, we evaluate our solution on the commonly used circuit for oblivious 128 bit AES encryption[3] [8]. This circuit is often used as benchmark in MPC implementations for boolean functions, due to its relatively random structure and large size. The idea is to encrypt a value known by Alice by using an encryption key known by Bob.

Here we are interested to compare our sequential and fine-grained implementations to the one described in [16]. The obtained results are shown in Table 2.

For a single AES implementation, macroblock parallelization cannot be used, anyway fine-grained parallelization guarantees better results than sequential implementation. To compare our solution to other implementations, we run Huang et al. code [16]

[3] Boolean circuit description kindly provided by Benny Pinkas, Thomas Schneider, Nigel P. Smart and Stephen C. Williams.

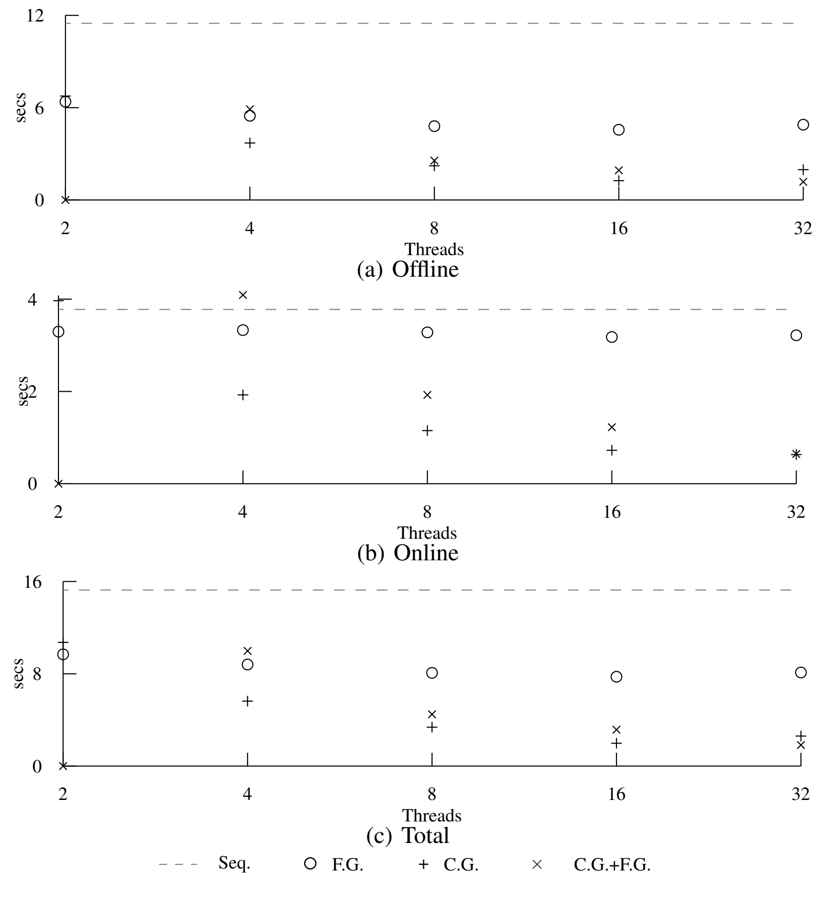

Fig. 4: Iris execution times.

	Phase	S	FG	Huang et al.
Offline	OT Prec.	0.001	0.001	0.540
	Garbling	0.133	0.082	0.898
	Garbled table Tx	0.039	0.044	
Online	Bob's secret Tx	0.000	0.000	0.038
	Alice's secret Tx	0.013	0.002	0.086
	Evaluation	0.066	0.017	0.311

Table 2: Runtimes (in seconds) of AES encryption protocol by using sequential implementation (S) and fine-grained (FG) parallelization. 8 cores have been used in parallel implementations. We run Huang et al. implementation in the same hardware used for our tests.

on the same computer used for our tests, obtaining results worse than ours. Again we can also observe that fine-grained parallelization offers better performance than serial execution.

To extend our analysis, we used the AES circuit as a block in a coarse grained parallelization of a circuit encrypting more than 128 bits provided by Alice by using the 128 bit encryption key of Bob, as shown in Figure 5.

For our tests we imagine to encrypt a gray-scaled image of size 256×256 pixels, hence $n = 4096$ AES encryption blocks are evaluated in parallel by using coarse-grained parallelization. Considering that associating a secret to an input available on the evaluator side is more expensive than associating a secret to an input available on the garbling side, Alice, having $256 \times 256 \times 8$ input bits, acts as garbler, whereas Bob, having 128 input bits, acts as evaluator. In Figure 6 we can observe the offline, online and total

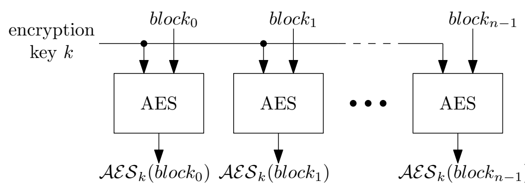
encryption key k
$block_0$
$block_1$
$block_{n-1}$
AES
AES
AES
$\mathcal{AES}_k(block_0)$
$\mathcal{AES}_k(block_1)$
$\mathcal{AES}_k(block_{n-1})$

Fig. 5: Two-party computation of 128-bit AES on large amount of data.

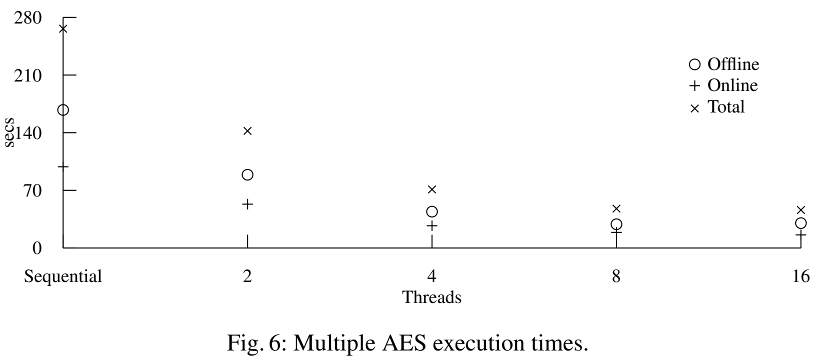

Fig. 6: Multiple AES execution times.

runtime of the protocol. As expected, parallel evaluation of AES blocks provides a significant improvement, confirming again that runtime decreases as $\approx 1/\mathit{threads}$ and by using 16 threads each AES takes less than 4ms online.

4.3 Security analysis

Parallel processing of single gates does not compromise the security of the protocol. As a matter of fact, the view of Alice and Bob in the fine-grained parallelization is equal to the one (except for the different order) obtained in the classical implementation. Coarse grained parallelization produces many GCs from a small number of description files that are combined to evaluate a more complex functionality. Again the view of Alice and Bob is equal to that obtained by evaluating a single larger garbled circuit in the common sequential implementation. Hence, being the security for Yao's protocol in the semi-honest model demonstrated in [22], also security of parallel implementations is granted.

5 Conclusions

In this paper we have shown that parallel processing can significantly improve the efficiency of Garbled Circuit technique in multi-party computation protocols. Two different types of parallelization have been proposed. Fine-grained parallelization allows the parallel evaluation of any garbled circuit processed to identify layers containing indipendent gates that can be evaluated concurrently. Coarse grained parallelization is based on the parallelization of macro-blocks and can be used whenever the same block is evaluated in parallel on different data.

The efficiency of the parallel implementations has been analyzed by addressing a biometric scenario, having an intrinsic parallel nature, and AES encryption. We demonstrated that both fine-grained and coarse-grained solutions provide significant runtime improvements. Macroblock parallelization is preferable when allowed by the intrinsic nature of the application, such as in a biometric identification scenario, otherwise gate parallelization can be used. The joint use of both techniques, by evaluating in parallel macroblocks, whose gates are still processed in parallel, results in a slight improvement.

Considering the results provided in this paper, efficient circuits for parallel GC evaluation could have different shapes with respect to circuits for classical sequential GC. By using coarse-grain parallelization a circuit designer is no more focused on the development of a whole optimized circuit, but to design blocks that can be evaluated in parallel, even if some gates can be superfluous. In fine-grained parallelization, even if reducing the number of non-XOR gates is still important, sometimes circuits with more gates can be evaluated more efficiently than others if they are characterized by a high level of parallelization. For example, having 8 threads available, the parallel evaluation of 4 gates can be more efficient than the sequential evaluation of only two gates. An accurate analysis of this issue is left for future research.

To extend our analysis, we are interested to apply our solutions to Garbled Circuits in a malicious setting and running the protocols on GPUs.

References

1. R. Agrawal and R. Srikant. Privacy-preserving data mining. *ACM Sigmod Record*, 29(2):439–450, 2000.
2. M. Barni, T. Bianchi, D. Catalano, R. Di Raimondo, R. Donida Labati, P. Failla, D. Fiore, R. Lazzeretti, V. Piuri, A. Piva, and F. Scotti. A Privacy-compliant Fingerprint Recognition System Based on Homomorphic Encryption and Fingercode Templates. In *Biometrics: Theory, Applications and Systems, 2010. BTAS 2010. IEEE Fourth International Conference on*, 2010.
3. M. Barni, T. Bianchi, D. Catalano, R. Di Raimondo, R. Donida Labati, P. Failla, D. Fiore, R. Lazzeretti, V. Piuri, A. Piva, and F. Scotti. Privacy-Preserving Fingercode Authentication. In *Multimedia and Security, 2010. MM&Sec 2010. 12th ACM Workshop on*, 2010.
4. M. Barni, P. Failla, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider. Privacy-preserving ecg classification with branching programs and neural networks. *Information Forensics and Security, IEEE Transactions on*, 6(2):452–468, 2011.
5. M. Barni, J. Guajardo, and R. Lazzeretti. Privacy Preserving Evaluation of Signal Quality With Application to ECG Analysis. In *Information Forensics and Security, 2010. WIFS 2010. Second IEEE International Workshop on*, 2010.
6. D. Beaver. Precomputing Oblivious Transfer. In *Advances in Cryptology – CRYPTO'95*, volume 963, pages 97–109, 1995.
7. M. Bellare and S. Micali. Non-interactive oblivious transfer and applications. In *Advances in Cryptology, CRYPTO'89 Proceedings*, pages 547–557. Springer, 1990.
8. J. Daemen and V. Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.
9. M. Deng, T. Bianchi, A. Piva, and B. Preneel. An efficient buyer-seller watermarking protocol based on composite signal representation. In *Proceedings of the 11th ACM workshop on Multimedia and security*, pages 9–18. ACM, 2009.
10. Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving face recognition. In *Privacy Enhancing Technologies*, pages 235–253. Springer, 2009.
11. Z. Erkin, A. Piva, S. Katzenbeisser, R.L. Lagendijk, J. Shokrollahi, G. Neven, and M. Barni. Protection and retrieval of encrypted multimedia content: when cryptography meets signal processing. *EURASIP Journal on Information Security*, 2007:17, 2007.
12. Z. Erkin, T. Veugen, T. Toft, and R.I. Lagendijk. Generating private recommendations efficiently using homomorphic encryption and data packing. *Information Forensics and Security, IEEE Transactions on*, 7(3):1053–1066, 2012.
13. T. K. Frederiksen and J. B. Nielsen. Fast and maliciously secure two-party computation using the gpu. Technical report, Cryptology ePrint Archive: Report 2013/046, 2013, 2013.
14. F. Hartung, Kalker T., and Lian S. *Digital Rights Management: Technology, Standards and Applications*, chapter Processing encrypted signals for DRM applications, (M. Barni, R. Lazzeretti, and C. Orlandi). CRC Press, 2013. To appear.
15. W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *ACM Computer and Communications Security (CCS'10)*, pages 451–462, 2010. `http://www.trust.rub.de/tasty/`.
16. Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011. `http://MightBeEvil.org`.
17. Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending Oblivious Transfers Efficiently. In *Advances in Cryptology – CRYPTO'03*, volume 2729, pages 145–161, 2003.
18. V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. How to combine homomorphic encryption and garbled circuits. In *Signal Processing in the Encrypted Domain–First SPEED Workshop–Lousanne*, page 100, 2009.

19. V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima. In *Cryptology and Network Security (CANS'09)*, volume 5888, pages 1–20, 2009.
20. R.L. Lagendijk, Z. Erkin, and M. Barni. Encrypted signal processing for privacy protection: Conveying the utility of homomorphic encryption and multiparty computation. *Signal Processing Magazine, IEEE*, 30(1):82 –105, Jan. 2013.
21. R. Lazzeretti, J. Guajardo, and M. Barni. Privacy Preserving ECG Quality Evaluation. In *Multimedia and security (MM&SEC), Proceedings of ACM workshop on*. ACM, 2012.
22. Y. Lindell and B. Pinkas. A Proof of Yao's Protocol for Secure Two-Party Computation. *Journal of Cryptology*, 22(2):161–188, 2009. Preliminary version at `http://eprint.iacr.org/2004/175`.
23. Y. Luo, S.C. Samson, T. Pignata, R. Lazzeretti, and M. Barni. An Efficient Protocol for Private Iris-Code Matching by Means of Garbled Circuits. In *Special Session on Emerging Topics in Cryptography and Image Processing, International Conference on Image Processing (ICIP)*, 2012.
24. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay – a secure two-party computation system. In *USENIX Security Symposium (Security'04)*, 2004. `http://www.cs.huji.ac.il/project/Fairplay`.
25. M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *ACM-SIAM Symposium On Discrete Algorithms (SODA'01)*, pages 448–457. Society for Industrial and Applied Mathematics, 2001.
26. NIST. US Department of Commerce, National Institute of Standards and Technology (NIST): Federal Information Processing Standard Publication 180-2, Announcing the SECURE HASH STANDARD, August 2002. `csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf`.
27. A. Paus, A.-R. Sadeghi, and T. Schneider. Practical secure evaluation of semi-private functions. In *Applied Cryptography and Network Security*, pages 89–106. Springer, 2009. `http://www.trust.rub.de/FairplaySPF`.
28. B. Pinkas, T. Schneider, N. P Smart, and S.C. Williams. Secure two-party computation is practical. In *Advances in Cryptology–ASIACRYPT 2009*, pages 250–267. Springer, 2009.
29. S. Pu, P. Duan, and J.-C. Liu. Fastplay–a parallelization model and implementation of smc on cuda based gpu cluster architecture. Technical report, Cryptology ePrint Archive, Report 2011/097, 2011., 2011.
30. A.C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, 1982.
31. A.C. Yao. How to Generate and Exchange Secrets. In *IEEE Symposium on Foundations of Computer Science (FOCS'86)*, pages 162–167, 1986.