RESEARCH ARTICLE

Logsum Using Garbled Circuits

José Portêlo[1,2]*, Bhiksha Raj[3], Isabel Trancoso[1,2]

1 INESC-ID, Lisbon, Portugal, **2** Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal, **3** Language Technologies Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, United States of America

* jose.portelo@inesc-id.pt

Abstract

Secure multiparty computation allows for a set of users to evaluate a particular function over their inputs without revealing the information they possess to each other. Theoretically, this can be achieved using fully homomorphic encryption systems, but so far they remain in the realm of computational impracticability. An alternative is to consider secure function evaluation using homomorphic public-key cryptosystems or Garbled Circuits, the latter being a popular trend in recent times due to important breakthroughs. We propose a technique for computing the logsum operation using Garbled Circuits. This technique relies on replacing the logsum operation with an equivalent piecewise linear approximation, taking advantage of recent advances in efficient methods for both designing and implementing Garbled Circuits. We elaborate on how all the required blocks should be assembled in order to obtain small errors regarding the original logsum operation and very fast execution times.

OPEN ACCESS

Citation: Portêlo J, Raj B, Trancoso I (2015) Logsum Using Garbled Circuits. PLoS ONE 10(3): e0122236. doi:10.1371/journal.pone.0122236

Academic Editor: Francesco Pappalardo, University of Catania, ITALY

Received: July 28, 2014

Accepted: December 19, 2014

Published: March 26, 2015

Data Availability Statement: All code necessary to replicate this study is available on GitHub via https://github.com/josemportelo/PLoS_ONE.

Funding: José Portêlo and Isabel Trancoso were supported by FCT grants SFRH/BD/71349/2010, PTDC/EIA-CCO/122542/2010 and UID/CEC/50021/2013. Bhiksha Raj was supported by NSF grant number 1017256. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing Interests: The authors have declared that no competing interests exist.

Introduction

An increasing number of server-based applications perform tasks such as classification, processing and analysis of user data. Often, these data are *private*, and should not be exposed to the server. Likewise, the server's inputs to these computations are also private and may not be exposed to the user. Hence, it becomes necessary to perform the computations in a *privacy preserving* manner, such that the user's data and the server's inputs are not revealed to one another, while yet ensuring that the appropriate party gets the correct result from the computations.

Stated in abstract terms, a user Alice possesses data $\mathbf{x}$ (e.g. email, a biometric measurement, or a voice sample). System Bob is capable of computing a function $f(\mathbf{x};\theta)$ with parameter θ (e.g. a spam filter or a biometric classifier). Alice desires to obtain $f(\mathbf{x};\theta)$ from Bob; however, she is unwilling to expose $\mathbf{x}$ to him. Bob too is unwilling to expose θ to Alice. The challenge of privacy-preserving computation is to enable Alice to obtain $f(\mathbf{x};\theta)$ such that Bob learns nothing of $\mathbf{x}$ and Alice learns nothing more of θ besides what she may glean from $f(\mathbf{x};\theta)$ itself.

In principle, such transactions could be achieved using fully homomorphic encryption [1], which permits computation to be performed on encrypted data. However, practical fully homomorphic encryption schemes remain computationally impractical in spite of recent advances [2, 3].

Instead, Alice and Bob must achieve their goals through secure function evaluation (SFE) [4, 5]. SFE involves repeated exchange of partial information between Alice and Bob using protocols which mask the exchanged values using a combination of techniques such as public-key encryption [6, 7], oblivious transfer (OT) [8], etc., until the appropriate party gets the desired result.

Arguably the most popular approach to SFE is to cast the function to be evaluated as a *garbled* Boolean circuit (GC) [4, 9, 10]. Like logic circuits, GCs are composed of logic gates, with each gate represented by a truth table. However, while in regular logic circuits the inputs and outputs of each truth table are binary values (0/1), in GCs each possible entry of the truth table is an encrypted value which can only be "unlocked" by a input-specific key pair. The entries, in turn, too are keys which may be used to unlock other gates downstream the circuit.

Although in principle any function could be computed in this manner, practical realizations face computational challenges. Evaluation of a GC requires exchange of information through OT and repeated decryption of gate outputs, both of which are highly expensive operations. Ideally the circuit itself must be designed to minimize this overhead.

Traditionally, circuit optimization has largely been viewed as the problem of minimizing the number of gates in the circuit. The problem of finding the smallest circuit to perform a given Boolean function is known to be intractable in general [11], although a number of effective algorithms have nevertheless been proposed [12–15].

In the case of GCs, we must also consider other factors. Not all gates are equally expensive to compute: gates such as NOT and XOR do not require explicit encryption in a GC and are orders of magnitude cheaper to compute than other gates, and must preferentially be used. Elsewhere, where feasible, complex operations may need to be replaced with cheaper approximations or or encrypted table lookups. GCs do not support looping operations. In addition, in situations where the underlying computation is floating-point, the effect of the loss of resolution from fixed-point implementation must also be considered.

The design of Garbled Circuits is thus a complex problem. A number of toolkits have been developed to support the conversion of algorithmic circuits to their garbled forms [16–19]. Typically, they include several optimized garbled components for "primitives", such as multiplication, maximum-value computation, vector inner-products, etc., which may be employed in the design of larger circuits. All of these primitives are *simple*, in the sense that they embody linear arithmetic computation over one or more inputs.

In this paper we develop a GC for a very important complex non-linear primitive, the *logsum* operation, a key building block in many statistical and signal processing computations. We build upon previous work [20, 21]. In [20] the authors use a GC to compute the logarithm as an integer operation, i.e., for input $a > 0$, the logarithm is computed as $b = \lfloor \log_2 a \rfloor + 1$. Because b is always an integer approximating the real value of the logarithm, this technique would lead to large errors if it were used for computing the logsum operation. In [21] the authors use a homomorphic encryption system to compute the logsum by successively performing secure protocols for exponentiation and logarithms. While this technique has reduced error with appropriate pre-scaling of the inputs, the encryption imposes large computational overhead and performing all the required protocols requires substantial amounts of time. In order to avoid the large computation errors or large execution times of the previous approaches, our design casts the logsum operation as a combination of simple operations such as table lookups, additions and multiplications, arranged to minimize the loss of resolution in the output while simultaneously retaining efficiency through the use of low-cost computational units, such that the actual garbled computation may be performed in reasonable time.

This paper starts with necessarily brief overviews of the different operations involved in logsum computations, and the basic idea underlying Garbled Circuit protocols. We propose a new method for computing the logsum operation using GCs, starting with an array of just two values, and later generalizing it to any number of input values. The performance of the proposed method is then analyzed for two different GC toolkits in terms of execution time for different numbers of input values, different lengths (in bits) of these values, and different operation parameters. The paper ends with some concluding remarks.

Methods

Logsum operation

In many signal processing and pattern classification problems it is necessary to compute the occurrence probability of an event X. The probability models employed are commonly mixture models of the form $P(X) = \sum_{i=1}^{N} P(X, Y_i)$. In order to avoid underflow/overflow problems, the actual computations are usually performed with *log* probabilities. Rather than computing $P(X)$, one works with $\log P(X)$, which is in turn expressed as in terms of the logarithm of the component probabilities as

$$z(m) = \log\left(\sum_{i=1}^{N} \exp(m_i)\right), \tag{1}$$

where $z(m) = \log(P(X))$, and $m_i = \log(P(X,Y_i))$. The equation above is a *logsum* operation. To further avoid overflow/underflow problems that arise when all the m_i terms become too large in magnitude, Equation 1 is usually further recast as

$$z(m) = m_X + \log\left(\sum_{i=1}^{N} \exp(m_i - m_X)\right), \tag{2}$$

with $m_X = \max_i m_i$.

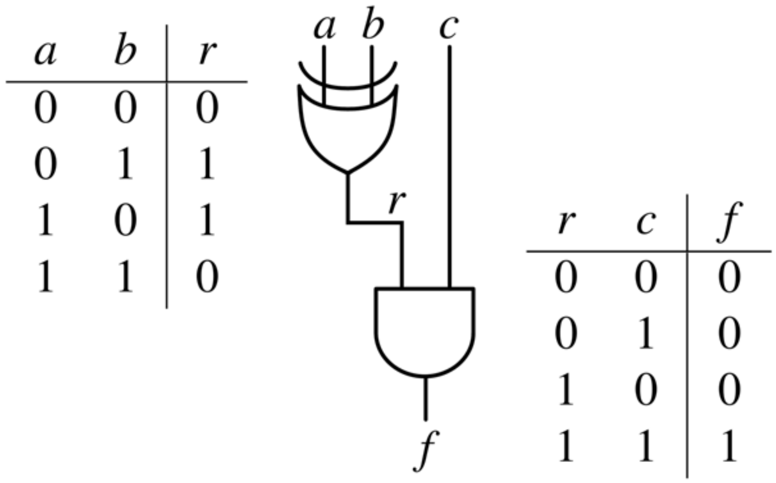
Logsum operations are found in a variety of signal processing and statistical computation tasks, such as sparse signal recovery [22], matrix recovery [23], and in the evaluation of a large variety of statistical models such as latent Dirichlet allocation [24] and Gaussian mixture models [25]. Thus, the logsum operation may in fact be considered one of the key building blocks in statistical and signal processing computations.

Garbled Circuits

Garbled Circuits are best explained with an example. Bob has Boolean input a and Alice has inputs b and c. They wish to compute the function $f(a,b,c) = (a \oplus b) \wedge c$. The logic circuit for $f(a,b,c)$ and the truth tables for the gates in it are shown in Fig. 1. However, Bob does not want to reveal a to Alice, and Alice does not want to reveal b or c to Bob. In order to still be able to compute $f(a,b,c)$, they must hence *garble* the circuit.

Bob generates the logic circuit implementing $f(\cdot)$. Thereafter, for each input and intermediate value (a, b, c, r in the example), Bob generates two private keys, one for each bit value, totaling eight keys in our example: $K_a^{0/1}$, $K_b^{0/1}$, $K_c^{0/1}$, $K_r^{0/1}$. For gates that generate intermediate values ($r = a \oplus b$ in our example), he replaces the outputs of the truth table with the encryption of the key corresponding to the output, performed with the keys corresponding to the inputs. For example, for $a = 0$, $b = 1$ the output is $r = 1$; the corresponding input keys are K_a^0, K_b^1 and the encrypted output is $\mathcal{E}_{K_a^0}(\mathcal{E}_{K_b^1}(K_r^1))$. For the output gate ($f = r \wedge c$ in our example) he encrypts the output value itself. The garbled values for our example are presented in Table 1.

To compute the function Bob transmits the keys corresponding to his bit choice for his inputs $K_a^?$ to Alice. Alice recovers the keys corresponding to the bit choice for her inputs $K_b^?$, $K_c^?$,



a	b	r
0	0	0
0	1	1
1	0	1
1	1	0

r	c	f
0	0	0
0	1	0
1	0	0
1	1	1

Fig 1. Example of a logic circuit and corresponding truth tables. The circuit implements an XOR gate ($r = a \oplus b$) followed by an AND gate ($f = r \wedge c$). The binary truth tables for the XOR and AND gates are presented on the left hand side and right hand side of the circuit, respectively.

doi:10.1371/journal.pone.0122236.g001

Table 1. Example of garbled truth tables.

a	*b*	*r*
K_a^0	K_b^0	$\mathcal{E}_{K_a^0}(\mathcal{E}_{K_b^0}(K_r^0))$
K_a^0	K_b^1	$\mathcal{E}_{K_a^0}(\mathcal{E}_{K_b^1}(K_r^1))$
K_a^1	K_b^0	$\mathcal{E}_{K_a^1}(\mathcal{E}_{K_b^0}(K_r^1))$
K_a^1	K_b^1	$\mathcal{E}_{K_a^1}(\mathcal{E}_{K_b^1}(K_r^0))$
r	***c***	***f***
K_r^0	K_c^0	$\mathcal{E}_{K_r^0}(\mathcal{E}_{K_c^0}(0))$
K_r^0	K_c^1	$\mathcal{E}_{K_r^0}(\mathcal{E}_{K_c^1}(0))$
K_r^1	K_c^0	$\mathcal{E}_{K_r^1}(\mathcal{E}_{K_c^0}(0))$
K_r^1	K_c^1	$\mathcal{E}_{K_r^1}(\mathcal{E}_{K_c^1}(1))$

[Left] Garbled table for the XOR gate ($r = a \oplus b$). The garbled values of the keys $K_r^{0/1}$ associated with output r, $\mathcal{E}_{K_a^{0/1}}(\mathcal{E}_{K_b^{0/1}}(K_r^{0/1}))$, are encrypted using the keys $K_a^{0/1}$ and $K_b^{0/1}$ associated with inputs a and b, respectively. [Right] Garbled table for the AND gate ($f = r \wedge c$). The garbled values of output f, $\mathcal{E}_{K_r^{0/1}}(\mathcal{E}_{K_c^{0/1}}(0/1))$, are encrypted using the keys $K_r^{0/1}$ and $K_c^{0/1}$ associated with inputs r and c, respectively.

doi:10.1371/journal.pone.0122236.t001

as well as the truth tables for the gates of the circuit from Bob using oblivious transfer [8]. Because of the properties of OT, Bob does not learn either the value of Alice's bits or the portion of the truth table she actually needs to perform her computations. To evaluate the circuit Alice successively evaluates each of the gates in the circuit. The nature of the computation is such that for each gate, Alice will possess two keys, one for each input. She decrypts all four encrypted output in the truth table for the gate, using her two keys. The keys will be inappropriate for three of the four outputs; hence Alice can only correctly decipher one of the four encrypted values, which will be a key for the next gate. After repeating this process for all gates, she finally obtains the desired value *f*. Alice and Bob never learn each others' inputs.

For a long time it was believed that GCs were of purely theoretical interest, but many advances have made GC much more efficient and practical to use. For instance, all the OTs for transferring the input ciphers may be pre-computed [26], meaning that all the computationally expensive operations regarding the OT are performed offline, and only an additional yet simple operation per OT is performed while evaluating the GC. Furthermore, all the OT may be computed efficiently by using shorter ciphers [27, 28], which can be implemented using elliptic curve cryptography [29]. In the gate decryption phase, the point and permute technique [16] may be used to reduce the total number of required decryptions from 4 to 1 by associating a permutation bit to the input ciphers chosen using OT. Finally, all the XOR gates of a GC may be evaluated for "free" [30], i.e., without significant computational cost, as the XOR gate does not need a garbled table and its evaluation consists of XOR-ing its garbled input values. NOT gates are similarly "free" as well. Custom designing circuit operations to preferentially utilize these gates will result in faster evaluation of the resulting Garbled Circuit [31].

On top of these implementation tricks, other useful properties of GC have been found. For example, it is possible to reuse the same GC several times under some conditions [32] by using functional encryption [33, 34], which means that if the same operation must be computed several times we only need a single circuit instead of one circuit for each time we want to compute that function. Moreover, several privacy and security guarantees regarding GC have been proven [9, 35–37].

In implementing the basic operations using GC, some issues different from the ones faced when using hardware circuits arise. Decision making operations such as comparisons (greater/lesser-than, maximum/minimum) and multiplexing can be implemented similarly to digital circuits. However, branching (if-then-else) and cycles/recursion (for loop, do-while) should be avoided. The first requires building and evaluating circuits for all possibilities in order to hide which one was actually chosen. The second requires explicitly expanding the cycles, one circuit per iteration, which may result in a huge intractable circuit. Another problem arises when computing non-linear functions, since the usual trick of computing the initial elements of the corresponding Taylor series is impractical to implement using GCs. An efficient work-around is to replace these operations with piecewise linear approximations (PLA), whose coefficients can be stored in look-up tables. An additional restriction exists on the number representation of the ciphered values. Operations on floating-point values usually require some normalization on them before they are performed, introducing an additional and unnecessary computational cost. To counter this, a fixed-point representation is preferred, despite a possible loss of precision in representing those same values.

Finally, given the wide-spread popularity of GCs, a huge variety of software implementations of all the required protocols and encryption systems are available. One of the first practical implementations of GC was Fairplay [16], and since then a number of different implementations have been developed in order to further improve their efficiency and practicality [17–19].

Logsum operation using Garbled Circuits

We now show how to perform the computation of the logsum of an array of values using GC. We start by describing how the logsum operation can be computed using a piecewise linear approximation, then we describe the full circuit for the simple situation where there are only two input values, and finally we generalize it to any number of input values.

Piecewise linear approximation. Many algorithms rely on evaluating non-linear functions in order to compute an output. These functions are normally replaced by the corresponding Taylor series, whose terms are computed until the error between iterations falls below a pre-determined threshold. However, when limited computational resources are available, the non-linear function is often be replaced by a PLA. A method for computing piecewise linear approximations using GCs has already been proposed [38], but it only focuses on a 1^{st}-order approach (ramp quantization (RQ)), discarding a 0^{th}-order approach (step quantization (SQ)). It is nevertheless instructive to evaluate both options in the context of GCs, where computation is at a premium. Although it is expected that the error obtained with the SQ approach should be larger than the one obtained with the RQ approach, the computational benefits of the lower-order approximation can outweigh the increased error from loss of resolution.

A standard way to implement a PLA is to split the domain of the input function $f(\cdot)$ into k intervals, and then compute linear approximations $f(\cdot)$ for each interval, adjusting all the interval limits t_i and linearization parameters $n_{1,i}$ and $n_{2,i}$ in order to obtain the minimum error possible regarding $f(\cdot)$. These parameters are then placed in the form of a look-up table such as the one presented in Table 2.

For obtaining the correct parameter pair $(n_{1,i}, n_{2,i})$ for each value of m, it is necessary to compare m with all the t in order to find an index j such that $t_j < m < t_{j+1}$, allowing for computing $f_{\mathrm{PLA}}(m) = n_{1,j+1} \cdot m + n_{2,j+1} \approx f(m)$, an approximation of the desired value. In the situations where $m < t_1$ or $m > t_{k-1}$, the approximation of the desired value is obtained by computing $f_{\mathrm{PLA}}(m) = n_{1,1} \cdot m + n_{2,1}$ or $f_{\mathrm{PLA}}(m) = n_{1,k} \cdot m + n_{2,k}$, respectively. This last scenario is not unlikely, since many functions have an unbounded domain, meaning that it is necessary to force artificial bounds when optimizing the locations of t. Therefore, special care is required while computing $n_{*,1}$ and $n_{*,k}$, so that they do not provide increasingly erroneous outputs as the difference between m and t_1 or t_{k-1} increases.

Higher order approximations. We did not consider any higher order polynomial approximations for the logsum operation for a variety of reasons. Let d be the degree of the approximation polynomial for the logsum operation. Preliminary experiments showed that although there was some small improvement in terms of the obtained error as d increased, the improvement rate decreased sharply. In particular, for $d \geq 3$ (cubic and higher order approximations) the error obtained is almost the same as for the situation where $d = 2$ (quadratic approximation). Also, increasing d leads to a quadratic increase in the number of required multiplications for evaluating the look-up table. Even if $d = 2$ is considered, this would require computing three multiplications instead of just one. Since the multiplication is by far the most computationally demanding operation in our GC approach, increasing d would lead to a significant and undesirable increase in the overall execution time. Finally, and most importantly, increasing d

Table 2. Look-up table with k entries for the piecewise linear approximation.

m:t	$f(m)$:$n_1 \cdot m + n_2$	
$m \leq t_1$	$n_{1,1}$	$n_{2,1}$
$t_1 < m \leq t_2$	$n_{1,2}$	$n_{2,2}$
...	...	...
$t_{k-2} < m \leq t_{k-1}$	$n_{1,k-1}$	$n_{2,k-1}$
$t_{k-1} < m$	$n_{1,k}$	$n_{2,k}$

Note that the domain of the i^{th} interval is given by $]t_{i-1}, t_i]$.

doi:10.1371/journal.pone.0122236.t002

leads to having to perform several multiplications in cascade, which, given a GC approach, causes an exponential increase of the number of bits required to represent the output.

Logsum of $N = 2$ values. Recalling Equation 2, the logsum of an array of numbers can be written as

$$z(m) = m_X + \log\left(\sum_{i=1}^{N} \exp(m_i - m_X)\right), \tag{3}$$

where $m_X = \max_i m_i$. For the simplest case where $N = 2$, this equation can be simplified to

$$z(m) = m_X + \log(1 + \exp(m_N - m_X)), \tag{4}$$

with $m_X = \max(m_1,m_2)$, $m_N = \min(m_1,m_2)$. After performing PLA, the approximation for the logsum operation is given by

$$z(m) \approx m_X + (n_1 \cdot (m_N - m_X) + n_2). \tag{5}$$

This is the equation implemented in our approach. Fig. 2 contains a circuit diagram showing how the logsum is computed using GC, with $N = 2$ inputs and $k = 4$ look-up table entries.

This diagram raises several implementation concerns. One of the most relevant is the number of bits (ℓ)—how it increases with each computation and what can be done to keep it as small as possible, since more bits mean more computational costs and consequently longer execution times. The operations where the number of bits of the output is larger than the number of bits of the inputs are ADD (1 extra bit), SUB (1 extra bit) and MUL (ℓ extra bits), so the other operations will be ignored in this analysis. Regarding the SUB operation, since the output will always be a negative number, the sign bit is uninformative, and therefore can be discarded. As for the MUL and the first ADD operations, the ℓ least significant bits of $f_{\text{PLA}}(m)$ need to be truncated, as otherwise one may end up with a scaling factor different from the one of m_X (recall that all numbers are being represented with fixed-point instead of floating-point). We also remove the most significant bit, as preliminary experiments suggest that it never contains any useful information, i.e., it is always 0. On the last ADD we do not remove any bits, thus increasing the total number of bits by 1 every time the logsum operation is computed.

A second concern is the choice between the SQ and RQ approaches to the PLA. Both of them were implemented following the diagram in Fig. 2, but whereas the RQ approach requires all the blocks, for the SQ approach the MUL and the first ADD block can be ignored, since in this situation $n_{1,?} = 0$. This also means that for the SQ approach there is no need to reduce the number of bits of the output.

Logsum of any N values. The Garbled Circuit described previously implements the logsum operation for the simplest case of $N = 2$ input values. Here we present the generalization of this methods to any given N. Let us start by recalling the original formulation of the logsum in Equation 3, which for $N = 4$ can be written as

$$\begin{aligned} z(m) &= \log(\exp(m_1) + \exp(m_2) + \exp(m_3) + \exp(m_4)) \\ &= \log\left(e^{\log(\exp(m_1)+\exp(m_2))} + e^{\log(\exp(m_3)+\exp(m_4))}\right) \\ &= \log\left(e^{z(m')} + e^{z(m'')}\right), \end{aligned} \tag{6}$$

with $m' = (m_1,m_2)$ and $m'' = (m_3,m_4)$. This means that the logsum for $N = 4$ input values can be obtained by computing two logsum operations, each of two input values. This process can be repeated as many times as needed, leading to a hierarchical tree structure for computing the logsum operation, as illustrated in Fig. 3. With this structure, the difference in the number of bits between the output and the inputs is $p = \lceil \log_2 N \rceil$.

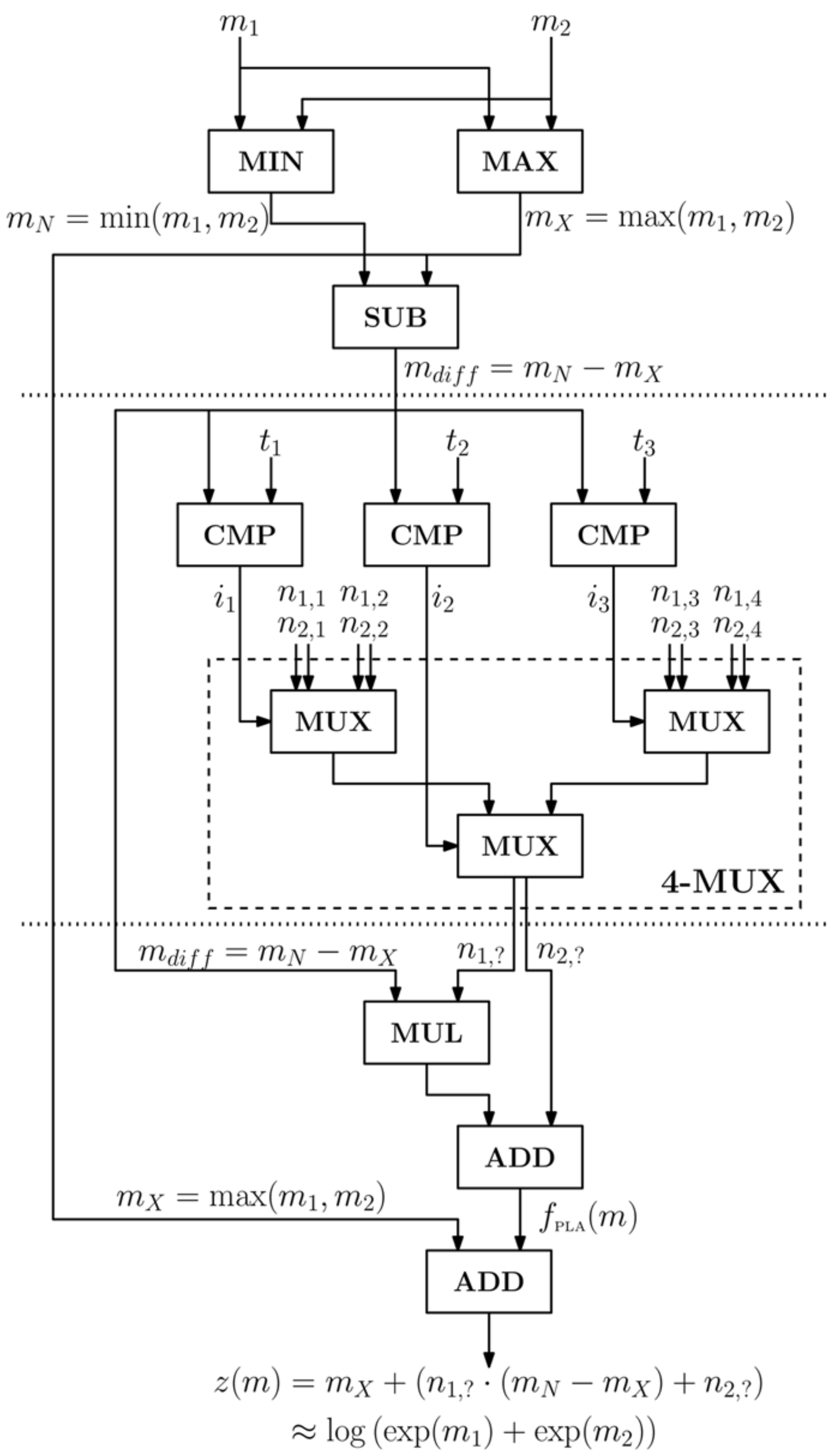


Fig 2. Circuit diagram for the logsum operation with N = 2 inputs and k = 4 look-up table entries. There are three major blocks in this diagram, separated from each other by horizontal dotted lines. In each block, the individual boxes correspond to a simple operation. The first block contains the computation of all the necessary elements for obtaining the logsum $z(m)$, namely m_X, m_N and $m_{diff} = m_N – m_X$, by means of computing a minimum (MIN), a maximum (MAX) and a subtraction (SUB), respectively. The second block

contains the circuit for obtaining the parameters for the PLA of the logsum operation. It starts by performing comparisons (CMP) between m_{diff} and the look-up table entries t, then the corresponding decisions i are used as control inputs to the hierarchical multiplexer (4-MUX), and finally the unknown piecewise linear approximation parameters $n_{1,?}$ and $n_{2,?}$ are outputted. The third block performs the linear approximation of the logsum operation, outputting the final result. It performs a multiplication (MUL) and an addition (ADD) between m_{diff} and $n_{1,?}$ and $n_{2,?}$, respectively. The final operation is an addition between m_X and $f_{\mathrm{PLA}}(m)$, thus completing the computation of the logsum.

doi:10.1371/journal.pone.0122236.g002

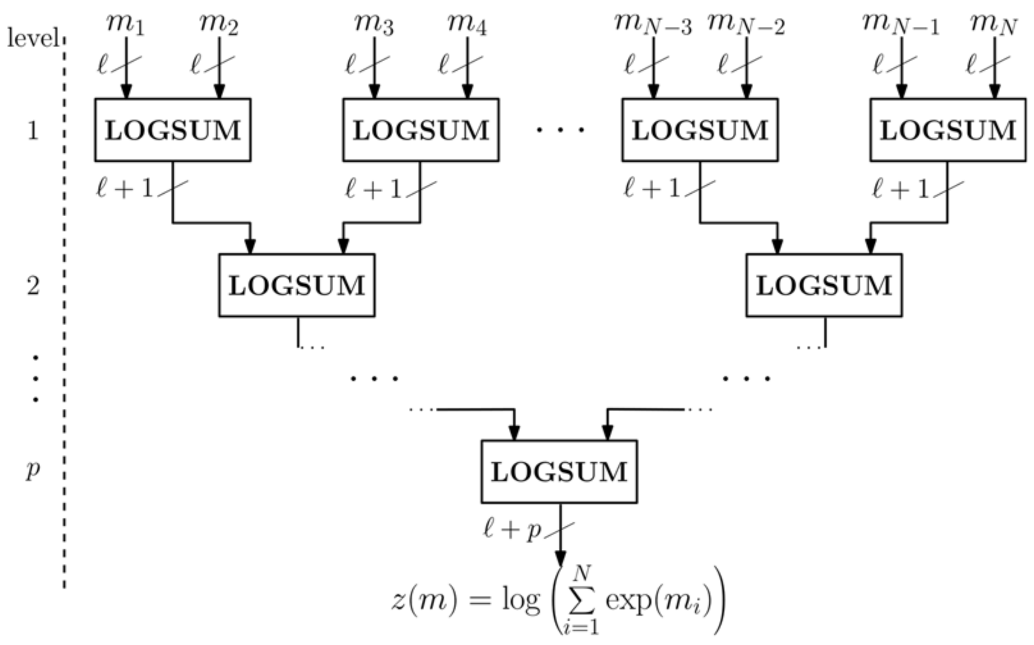


Fig 3. Circuit diagram for the logsum operation for any N inputs. The logsum of N elements is computed using a hierarchical structure, computing the logsum of two elements at a time. After each level the number of bits required to represent the output increases by 1. If the inputs are represented by ℓ bits, the output will be represented by $\ell+p$ bits, with $p = \lceil \log_2 N \rceil$.

doi:10.1371/journal.pone.0122236.g003

Results and Discussion

In order to evaluate the performance of our implementation of the logsum operation, two sets of experiments were performed. The first set was designed to analyze the effect in the absolute error of changing the number of bits of the input (ℓ) and the number of entries on the look-up table (k) when computing the logsum of $N = 2$ input values. The second set was designed to analyze the effect in terms of absolute error and execution time of computing the logsum of input arrays of different sizes N, for selected values of ℓ and k. We considered two different GC toolkits: the one from the VIPP group [18] and the JustGarble toolkit [19]. Both toolkits implement many techniques mentioned earlier for improving the efficiency of GC, namely pre-computing OT, using the permute technique for gate decryption, free XOR gate evaluation, etc. However, the VIPP toolkit was designed for performing tasks where only a very small number of gates (i.e. a few thousand) are required and not-so-small execution times (i.e. a few seconds) are acceptable. Examples of such tasks include the approaches for iris matching and ECG signal quality evaluation, described in [39] and [20], respectively. Therefore, the VIPP toolkit was not optimized in terms of circuit file size and memory usage. As for the JustGarble toolkit, it was designed in order to both maximize the number of gates that can be used in a single circuit and minimize execution time and memory usage. This means that special care was taken in terms of memory management during the software implementation phase, which has a profound impact in terms of the overall execution time. The advantage of the VIPP toolkit is that, unlike the JustGarble toolkit, it comes with efficient basic block designs in terms of the amount of non-XOR gates for most of the operations we require for computing the logsum. The errors obtained in all experiments are always the same regardless of the GC toolkit considered; the difference occurs only when the execution times are analyzed.

The results obtained in terms of error in the first experiment for both the step quantization (SQ) and the ramp quantization (RQ) approaches are presented in Tables 3 and 4, respectively. In every cell of both tables, the first and second rows contain the mean and maximum absolute error with respect to the true value of the logsum, respectively. Each experiment was performed by generating one million uniformly distributed pairs of numbers and computing the logsum over each pair, both with our approach and the exact equation. Analyzing the results for the SQ approach, we notice that the error obtained depends much less on ℓ than on k, as increasing the number of bits beyond $\ell = 12$ does not visibly reduce the error, but increasing k leads to a significant reduction of the error, specially for larger values of ℓ. This is expected, since the maximum error in this approach is directly governed by the size of the steps, which is much greater than the precision of the values. In the RQ approach, both increasing ℓ and k leads to large reductions in the error. In some cases, even small changes in the parameters result in differences of up to one order of magnitude. Although the error obtained using the SQ approach may be expected to always be larger than the one obtained using the RQ approach, for $\ell = 8$ this is not the case. The reason may be that the effect of removing the least significant bits after the MUL block is more significant if there is a small number of bits to start with.

The results obtained in terms of error in the second experiment for the SQ and RQ approaches are presented in Tables 5 and 6, respectively. Similarly to the previous experiment, in every cell of all the tables the first and second rows contain the mean and maximum absolute error regarding the logsum, respectively, and one million uniformly distributed arrays of size N

Table 3. Absolute error for different values of ℓ and **k**; step quantization (SQ) approach.

	k		
ℓ	**32**	**64**	**128**
8	1.17×10^{-2}	1.01×10^{-2}	1.03×10^{-2}
	(5.49×10^{-2})	(3.74×10^{-2})	(4.05×10^{-2})
12	0.53×10^{-2}	0.28×10^{-2}	0.15×10^{-2}
	(2.74×10^{-2})	(1.52×10^{-2})	(0.89×10^{-2})
16	0.52×10^{-2}	0.27×10^{-2}	0.13×10^{-2}
	(2.68×10^{-2})	(1.46×10^{-2})	(0.83×10^{-2})
24	0.52×10^{-2}	0.27×10^{-2}	0.13×10^{-2}
	(2.68×10^{-2})	(1.45×10^{-2})	(0.82×10^{-2})
32	0.52×10^{-2}	0.27×10^{-2}	0.13×10^{-2}
	(2.68×10^{-2})	(1.45×10^{-2})	(0.82×10^{-2})

ℓ corresponds to the number of bits representing the inputs and k is the number of entries of the look-up table for the piecewise linear approximation. In each cell, the first and second rows contain the mean and maximum absolute error with respect to the true value of the logsum, respectively.

doi:10.1371/journal.pone.0122236.t003

Table 4. Absolute error for different values of ℓ and k; ramp quantization (RQ) approach.

	k		
ℓ	**32**	**64**	**128**
8	1.69×10^{-2}	1.66×10^{-2}	1.57×10^{-2}
	(5.02×10^{-2})	(5.42×10^{-2})	(5.01×10^{-2})
12	1.10×10^{-3}	1.01×10^{-3}	1.02×10^{-3}
	(3.58×10^{-3})	(3.33×10^{-3})	(3.50×10^{-3})
16	1.45×10^{-4}	0.77×10^{-4}	0.68×10^{-4}
	(4.40×10^{-4})	(2.66×10^{-4})	(2.30×10^{-4})
24	1.20×10^{-4}	0.30×10^{-4}	0.08×10^{-4}
	(2.54×10^{-4})	(0.66×10^{-4})	(0.20×10^{-4})
32	1.20×10^{-4}	0.30×10^{-4}	0.07×10^{-4}
	(2.54×10^{-4})	(0.66×10^{-4})	(0.19×10^{-4})

ℓ corresponds to the number of bits representing the inputs and *k* is the number of entries of the look-up table for the piecewise linear approximation. In each cell, the first and second rows contain the mean and maximum absolute error with respect to the true value of the logsum, respectively.

doi:10.1371/journal.pone.0122236.t004

Table 5. Absolute error for selected values of N, ℓ and k; step quantization (SQ) approach.

N	**8**	**32**	**128**	**512**
ℓ = 8, *k* = 32	0.17×10^{-1}	0.24×10^{-1}	0.41×10^{-1}	0.66×10^{-1}
	(1.00×10^{-1})	(1.14×10^{-1})	(1.22×10^{-1})	(1.41×10^{-1})
ℓ = 12, *k* = 128	0.27×10^{-2}	0.35×10^{-2}	0.40×10^{-2}	0.46×10^{-2}
	(1.69×10^{-2})	(1.80×10^{-2})	(2.13×10^{-2})	(2.32×10^{-2})
ℓ = 24, *k* = 128	0.26×10^{-2}	0.32×10^{-2}	0.35×10^{-2}	0.37×10^{-2}
	(1.39×10^{-2})	(1.79×10^{-2})	(1.84×10^{-2})	(2.06×10^{-2})

N corresponds to the number of inputs for the logsum, ℓ corresponds to the number of bits representing the inputs and *k* is the number of entries of the look-up table for the piecewise linear approximation. In each cell, the first and second rows contain the mean and maximum absolute error with respect to the true value of the logsum, respectively.

doi:10.1371/journal.pone.0122236.t005

Table 6. Absolute error for selected values of N, ℓ and k; ramp quantization (RQ) approach.

N	**8**	**32**	**128**	**512**
ℓ = 8, *k* = 32	0.48×10^{-1}	0.79×10^{-1}	1.11×10^{-1}	1.43×10^{-1}
	(1.12×10^{-1})	(1.37×10^{-1})	(1.66×10^{-1})	(1.90×10^{-1})
ℓ = 12, *k* = 128	0.28×10^{-2}	0.48×10^{-2}	0.68×10^{-2}	0.89×10^{-2}
	(0.69×10^{-2})	(0.83×10^{-2})	(1.01×10^{-2})	(1.20×10^{-2})
ℓ = 24, *k* = 128	1.10×10^{-5}	1.40×10^{-5}	1.80×10^{-5}	2.40×10^{-5}
	(4.50×10^{-5})	(5.40×10^{-5})	(6.50×10^{-5})	(7.00×10^{-5})

N corresponds to the number of inputs for the logsum, ℓ corresponds to the number of bits representing the inputs and *k* is the number of entries of the look-up table for the piecewise linear approximation. In each cell, the first and second rows contain the mean and maximum absolute error with respect to the true value of the logsum, respectively.

doi:10.1371/journal.pone.0122236.t006

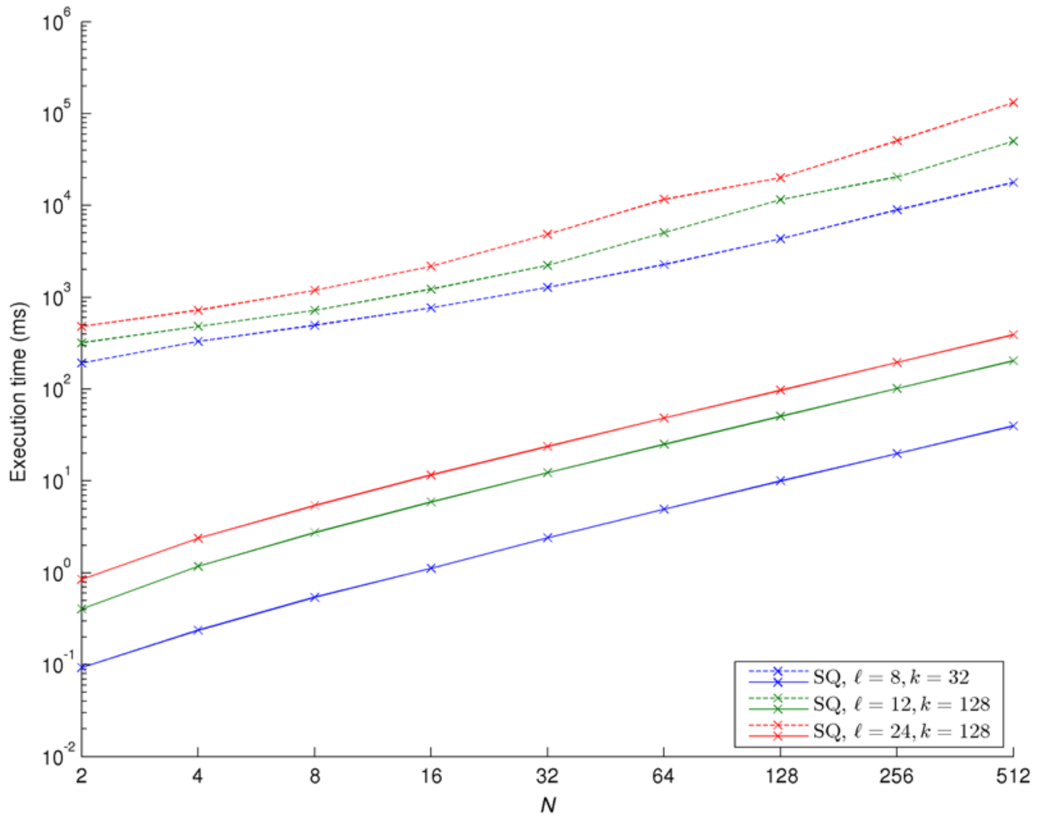


Fig 4. Execution time for selected values of N, ℓ and k, step quantization (SQ) approach, when an Intel Core i7-3630QM CPU @ 2.40GHz with a 6MB L3 cache memory and an 8GB DDR3 RAM memory is considered. *N* corresponds to the number of inputs for the logsum, *ℓ* corresponds to the number of bits representing the inputs and *k* is the number of entries of the look-up table for the piecewise linear approximation. The dashed lines correspond to the experiments performed with the VIPP toolkit; the solid lines correspond to the experiments performed with the JustGarble toolkit.

doi:10.1371/journal.pone.0122236.g004

were considered. Regarding the SQ approach, we observe that increasing *N* leads to only slightly increasing the error accumulated by successively computing the logsum, meaning that this technique scales well with *N* for all values of ℓ and *k*. Observing the results for the RQ approach, we notice that it only scales efficiently with *N* for larger values of ℓ and *k*. However, since this is the most interesting situation because it is the one where smaller errors are obtained, scenarios with small ℓ and *k* may be ignored.

We now analyze the execution time of our approach using GC. Currently, it takes a few nanoseconds for a microprocessor to compute a logsum over $N = 2$ elements. Because of the additional operations required by GC (encryption/decryption, shuffling, etc.), our approach requires substantially larger amounts of time to perform the logsum compared to the non-private counterpart. Nevertheless, for many real-time applications, this approach is still fast enough. Taking advantage of all techniques for efficient implementation of GC, all steps but the actual circuit evaluation can be performed offline; therefore, we only present the execution times for that particular step. We confirmed experimentally that if these techniques were not considered, the overall execution times would increase at least by one order of magnitude.

We considered two different processor and memory settings for our experiments: 1) an Intel Core i7-3630QM CPU @ 2.40GHz with a 6MB L3 cache memory and an 8GB DDR3 RAM memory and 2) an Intel Xeon E5530 CPU @ 2.40GHz with a 8MB L3 cache memory and a 48GB DDR3 RAM memory. The results obtained for the SQ and RQ approaches for the first

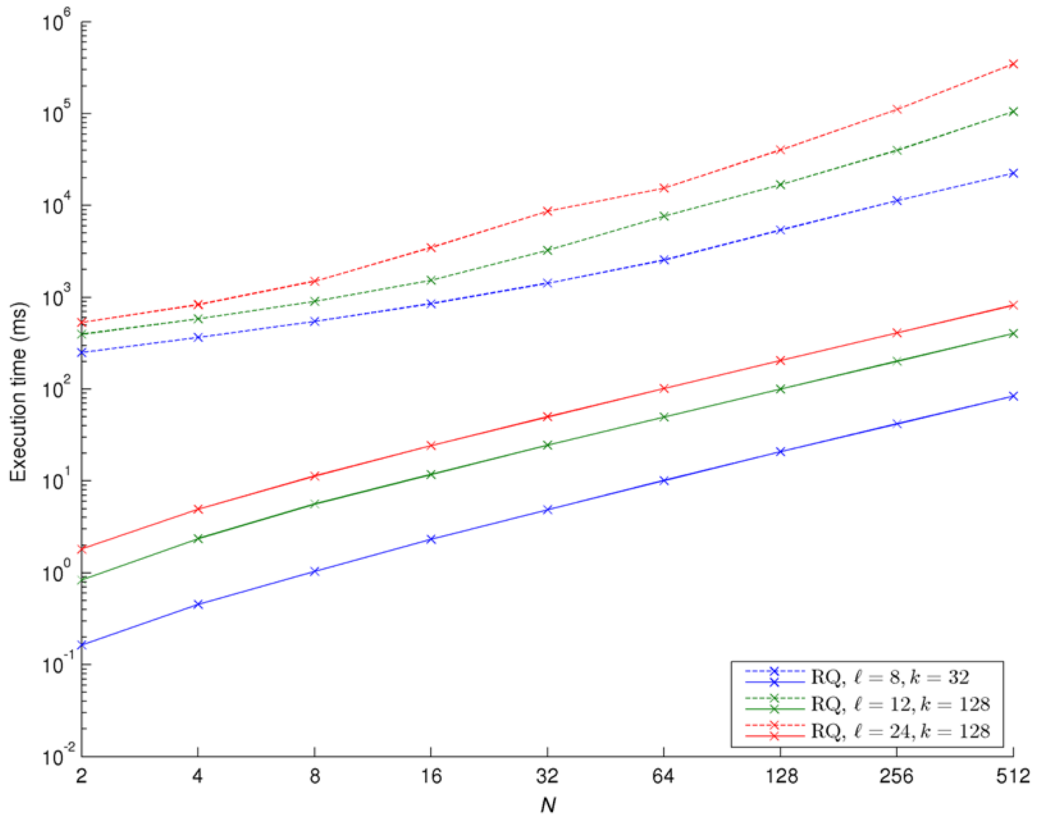


Fig 5. Execution time for selected values of N, ℓ and k, ramp quantization (RQ) approach, when an Intel Core i7-3630QM CPU @ 2.40GHz with a 6MB L3 cache memory and an 8GB DDR3 RAM memory is considered. *N* corresponds to the number of inputs for the logsum, ℓ corresponds to the number of bits representing the inputs and *k* is the number of entries of the look-up table for the piecewise linear approximation. The dashed lines correspond to the experiments performed with the VIPP toolkit; the solid lines correspond to the experiments performed with the JustGarble toolkit.

doi:10.1371/journal.pone.0122236.g005

setting are presented in Figs. 4 and 5, respectively. The execution times obtained for both GC toolkits when the second setting was considered are around 10% to 15% slower than the ones from the first setting, but have otherwise a very similar behavior to them. Therefore, and in order to avoid overloading this section, we decided not to reproduce them. Each of the results was obtained by averaging the execution time of one thousand consecutive runs of the logsum algorithm. As expected, for both GC toolkits and using the same values of ℓ and *k*, the RQ approach takes longer than the SQ approach to compute a logsum. We observe that for the VIPP toolkit there is a near quadratic increase of the execution time with increasing values of *N*, probably a consequence of the successively larger circuits that need to be loaded into memory. We also notice that for the JustGarble toolkit there is a constant linear relationship between how long it takes to compute the logsum and the number of inputs for all values of ℓ and *k*, meaning that our approach also scales nicely with *N* regarding the execution time. Finally and more importantly, the JustGarble toolkit is consistently two or three orders of magnitude faster than the VIPP toolkit, definitely showing that when larger garbled circuits are considered, it is extremely important to use a toolkit specially designed for optimal memory management.

Conclusions

This paper presents a technique for computing the logsum operation using Garbled Circuits. We describe a simple technique that approximates the logsum with a piecewise linear approximation, which is then expressed in the form of a garbled circuit that can be computed efficiently, while maintaining the privacy of its inputs. Furthermore, even within this format, unlike previous approaches, the solution is scalable, in the sense that the accuracy of the approximation can be traded off for computation time by increasing the parameters ℓ and k. The SQ and RQ options offer the following choice—the SQ approach can be used for fast computation where the underlying problem can tolerate lower precision, and RQ can be used where greater precision is required. Even with the increased computation, this method is still much faster than previously proposed homomorphic encryption approaches. We evaluated two different GC toolkits designed for distinctive ends, and concluded that memory management should be a prime concern when implementing a general purpose GC toolkit.

The logsum operation is an important cornerstone for many signal processing applications. The technique proposed here presents an ideal primitive to be incorporated in the implementation of such applications in a privacy-preserving scenarios.

Author Contributions

Conceived and designed the experiments: JP BR IT. Performed the experiments: JP. Analyzed the data: JP BR IT. Contributed reagents/materials/analysis tools: JP BR IT. Wrote the paper: JP BR IT.

References

1. Gentry C. A Fully Homomorphic Encryption Scheme. Ph.D. Thesis, Stanford University. 2009. Available: http://crypto.stanford.edu/craig/craig-thesis.pdf.
2. Coron J-S, Mandal A, Naccache D, Tibouchi M. Fully Homomorphic Encryption over the Integers with Shorter Public Keys. Advances in Cryptology—CRYPTO 2011. Springer LNCS. 2011; 6841: 487–504.
3. van Dijk M, Gentry C, Halevi S, Vaikuntanathan V. Fully Homomorphic Encryption over the Integers. Advances in Cryptology—EUROCRYPT 2010. Springer LNCS. 2010; 6110: 24–43.
4. Yao A. Protocols for Secure Computation (Extended Abstract). FOCS 1982: Proceedings of the IEEE Symposium on Foundations of Computer Science; 1982 Nov 3–5; Chicago, IL, USA. 1982. p.160–164. doi: 10.1109/SFCS.1982.38
5. Lindell Y, Pinkas B. Secure Multiparty Computation for Privacy-Preserving Data Mining. Journal of Privacy and Confidentiality. 2009; 1(1): 59–98.
6. ElGamal T. A Public Key Cryptosystem and a Signature Scheme based on Discrete Logarithms. IEEE Trans Inf Theory. 1985; 31(4): 469–472. doi: 10.1109/TIT.1985.1057074
7. Paillier P. Public-key Cryptosystems based on Composite Degree Residuosity Classes. Advances in Cryptology—EUROCRYPT'99. Springer LNCS. 1999; 1592: 223–238.
8. Naor M, Pinkas B. Oblivious Transfer and Polynomial Evaluation. STOC 1999: Proceedings of the ACM Symposium on Theory of Computing. 1999 May 1–4; Atlanta, GA, USA. 1999. p. 245–254. doi: 10.1145/301250.301312
9. Bellare M, Hoang V, Rogaway P. Foundations of Garbled Circuits. CCS 2012: Proceedings of the ACM Conference on Computer and Communications Security. 2012 Oct 16–18; Raleigh, NC, USA. 2012. p. 784–796. doi: 10.1145/2382196.2382279
10. Lazzeretti R, Barni M. Private Computing with Garbled Circuits [Applications Corner]. IEEE Signal Process Mag. 2013; 30(2): 123–127. doi: 10.1109/MSP.2012.2230540
11. Kabanets V, Cai JY. Circuit Minimization Problem. STOC 2000: Proceedings of the ACM Symposium on Theory of Computing. 2000 May 21–23; Portland, OR, USA. 2000. p. 73–79. doi: 10.1145/335305.335314
12. Karnaugh M. The Map Method for Synthesis of Combinational Logic Circuits. Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics. 1953; 72(5): 593–599.
13. McCluskey E. Detection of Group Invariance or Total Symmetry of a Boolean Function. Bell System Technical Journal. 1956; 35(6): 1445–1453. doi: 10.1002/j.1538-7305.1956.tb03836.x
14. Rudell R. Multiple-Valued Logic Minimization for PLA Synthesis. Memorandum No. UCB/ERL M86/65. University of California. 1986. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1986/ERL-86-65.pdf.

15. de Micheli G. Synthesis and Optimization of Digital Circuits. McGraw-Hill Higher Education. 1994.
16. Malkhi, D, Nisan N, Pinkas, B, Sella, Y. Fairplay—A Secure Two-Party Computation System. USENIX 2004: Proceedings of the USENIX Security Symposium. 2004 9–13 Aug; San Diego, CA, USA. 2004. p. 287–302.
17. Huang, Y, Evans, D, Katz, J, Malka, L. Faster Secure Two-Party Computation using Garbled Circuits. USENIX 2011: Proceedings of the USENIX Security Symposium. 2011 8–12 Aug; San Francisco, CA, USA. 2011. p. 539–554. Software available: http://mightbeevil.com/framework.
18. Pignata T. Garbled Circuit Designer and Executer from the Visual Information Processing and Protection (VIPP). Software available: http://clem.dii.unisi.it/~vipp/index.php/software/135-garbledcircuit.
19. Bellare, M, Hoang, V, Keelveedhi, S, Rogaway, P. Efficient Garbling from a Fixed-Key Blockcipher. SP 2013: Proc IEEE Symp Secur Priv. 2013 May 19–22; Berkeley, CA, USA. 2013. p. 478–492. doi: 10.1109/SP.2013.39. Software available: http://cseweb.ucsd.edu/groups/justgarble/.
20. Barni, M, Guajardo, J, Lazzeretti, R. Privacy Preserving Evaluation of Signal Quality with Application to ECG Analysis. WIFS 2010: Proceedings of the IEEE International Workshop on Information Forensics and Security. 2010 Dec 12–15; Seattle, WA, USA. 2010. p. 1–6. doi: 10.1109/WIFS.2010.5711460
21. Pathak M, Rane S, Sun W, Raj B. Privacy Preserving Probabilistic Inference with Hidden Markov Models. ICASSP 2011: Proc IEEE Int Conf Acoust Speech and Signal Process. 2011 May 22–27; Prague, Czech Republic. 2011. p. 5868–5871.
22. Shen Y, Fang J, Li H. Exact Reconstruction Analysis of Log-Sum Minimization for Compressed Sensing. IEEE Signal Process Lett. 2013; 20(12): 1223–1226. doi: 10.1109/LSP.2013.2285579
23. Dend Y, Dai Q, Liu R, Zhang S, Hu S. Low-Rank Structure Learning via Nonconvex Heuristic Recovery. IEEE Trans Neural Netw Learn Syst. 2013; 24(3): 383–396. doi: 10.1109/TNNLS.2012.2235082
24. Blei DM, Ng AY, Jordan MI, Lafferty J. Latent Dirichlet Allocation. J Mach Learn Res. 2003; 3: 993–1022.
25. Reynolds DA, Rose RC. Robust Text-Independent Speaker Identification using Gaussian Mixture Speaker Models. IEEE Transactions on Speech and Audio Processing. 1995; 3(1): 72–83. doi: 10.1109/89.365379
26. Beaver D. Precomputing Oblivious Transfer. Advances in Cryptology–CRYPTO'95. Springer LNCS. 1995; 963: 97–109.
27. Aiello B, Ishai Y, Reingold O. Priced Oblivious Transfer: How to Sell Digital Goods. Advances in Cryptology—EUROCRYPT 2001. Springer LNCS. 2001; 2045: 119–135.
28. Naor, M, Pinkas, B. Efficient Oblivious Transfer Protocols. SODA 2001: Proceedings of the Annual Symposium on Discrete Algorithms. 2001 Jan 7–9; Washington, DC, USA. 2001. p. 448–457.
29. Koblitz N. Elliptic Curve Cryptosystems. Math Comput. 1987; 48(177): 203–209. doi: 10.1090/S0025-5718-1987-0866109-5
30. Kolesnikov, V, Schneider, T. Improved Garbled Circuit: Free XOR Gates and Applications. ICALP 2008: Proceedings of the International Colloquium on Automata, Languages and Programming. 2008 Jul 6–13; Reykjavik, Iceland, 2008. p. 486–498. doi: 10.1007/978-3-540-70583-3_40
31. Kolesnikov, V, Sadeghi A-R, Schneider, T. Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima. CANS 2009: Proceddings of the International Conference on Cryptology and Network Security. 2009 Dec 12–14; Kanazawa, Japan. 2009. p. 1–20. doi: 10.1007/978-3-642-10433-6_1
32. Goldwasser, S, Kalai, Y, Popa, R, Vaikuntanathan, V, Zeldovich, N. Reusable Garbled Circuits and Succinct Functional Encryption. STOC 2013: Proceedings of the ACM Symposium on Theory of Computing. 2013 Jun 1–4; Palo Alto, CA, USA. 2013. p. 555–564. doi: 10.1145/2488608.2488678
33. Sahai A, Waters B. Fuzzy Identity-based Encryption. Advances in Cryptology—EUROCRYPT 2005. Springer LNCS. 2005; 3494: 457–473.
34. Boneh, D, Sahai, A, Waters, B. Functional Encryption: Definitions and Challenges. TCC 2011: Proceedings of the Theory of Cryptography Conference. 2011 Mar 28–30; Providence, RI, USA. 2011. p. 253–273. doi: 10.1007/978-3-642-19571-6_16
35. Boyd S, Kim S-J, Patil D, Horowitz M. Digital Circuit Optimization via Geometric Programming. Oper Res. 2005; 53(6): 899–932. doi: 10.1287/opre.1050.0254
36. Järvinen, K, Kolesnikov, V, Sadeghi A-R, Schneider, T. Garbled Circuits for Leakage-Resilience: Hardware Implementation and Evaluation of One-Time Programs. CHES 2010: Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems. 2010 Aug 17–20; Santa Barbara, CA, USA. 2010. p. 383–397. doi: 10.1007/978-3-642-15031-9_26
37. Mohassel P, Riva B. Garbled Circuits Checking Garbled Circuits: More Efficient and Secure Two-Party Computation. Advances in Cryptology—CRYPTO 2013. Springer LNCS. 2013; 8043: 36–53.

38. Pignata, T, Lazzeretti, R, Barni, M. General Function Evaluation in a STPC Setting Via Piecewise Linear Approximation. WIFS 2012: Proceedings of the IEEE International Workshop on Information Forensics and Security. 2012 Dec 2–5; Costa Adeje, Tenerife, Spain. 2012. p. 55–60. doi: 10.1109/WIFS.2012.6412625

39. Luo Y, Cheung S, Pignata T, Lazzeretti R, Barni M. An Efficient Protocol for Private Iris-code Matching by Means of Garbled Circuits. ICIP 2012: Proc Int Conf Image Proc. 2012 Sep 30–Oct 3; Lake Buena Vista, Orlando, FL, USA. 2012. p. 2653–2656.