CYBERNETICA
Institute of Information Security

Yao Garbled Circuits in Secret Sharing-based Secure Multi-party Computation

Oleg Šelajev, Sven Laur

T-4-15 / 2011

Oleg Šelajev[1,2], Sven Laur[2].
[1] ZeroTurnaround,
[2] University of Tartu, Institute of Computer Science

The research reported here was supported by:

1. the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS

Cybernetica research reports are available online at
`http://research.cyber.ee/`

Mailing address:
AS Cybernetica
Akadeemia tee 21
12618 Tallinn
Estonia

Yao Garbled Circuits in Secret Sharing-based Secure Multi-party Computation

Oleg Šelajev, Sven Laur

December 23, 2011

Abstract

Yao Garbled Circuits are a known solution for the secure multi-party computation problem. However, it's poor performance prevents the use of garbled circuits in a real life applications. In this report we propose a protocol for secure multi-party computation that uses Yao Garbled Circuits for Sharemind. Subtle details of existing Sharemind protocols allow us to implement essential parts of the Yao Circuit Evaluation (YCE) differently from general implementations. This approach could lead to a better performance result of YCE and offer a general way of adding new internal functionality of Sharemind.

1 Introduction

Sharemind is a data processing system capable of performing computations on input data without compromising its privacy. For an external Sharemind user, it looks like a virtual machine, that can operate on the data, but internally, Sharemind is a distributed system of so called 'miners'. A miner is just an independent machine executing Sharemind protocols. Sharemind uses secret sharing to preserve the privacy of the data. This means that before external data is imported into Sharemind system, it must be additively shared into three shares. Each share then goes to a respective miner. Sharemind implements a number of secure multi-party protocols for processing this information. As a result, the miner can produce meaningful statistics from data without sacrificing its privacy.

Secret sharing is one of the ways to ensure security of the computation. This is, actually, widely known and a lot of research has been done to discover possibilities to compute while preserving the privacy of the data. General ways to do that were known since 1982, when A.C.C.Yao proposed a way how to use boolean circuits to guard computation against the adversary who will behave honestly, but investigate information available to find other parties inputs [1].

We want to add functionality for using Yao Garbled circuits to the Sharemind framework. This will enrich Sharemind's choice of tools for performing secure multi-party computations. In this work we describe and implement a way to generate and evaluate garbled circuits in the

Sharemind environment. We present a way for specifying arbitrary binary operation on shared values between two miners. The evaluation of this operation will preserve the privacy of the arguments, thus providing a general way to enhance Sharemind's ability for privacy preserving computations.

2 Motivation

Currently, Sharemind uses a dozen of protocols for basic math functions. There are protocols for basic arithmetic, bit extracting routines, comparisons like greater-than, greater-than or equal and matrix multiplication. These low-level protocols are written in C++ and ensure Sharemind's ability to perform secure multi-party computation.

There are two problems with this approach. First of all, writing cryptographically safe protocols in C++ is not a trivial task, it is error-prone and generally difficult. So when anyone needs to add something to this low-level functionality, the current Sharemind architecture does not provide an easy way for so.

Another problem arises from the fact that not all math functions can be efficiently computed on the additively shared values. For example floating point math offers particular interest for data analysis. Usually, a floating point number is represented in a scientific notation with exponent and significand concatenated in a single 32-bit (or 64-bit) value. The additive sharing of such value loses the meaning of exponent and significand, so further operations on shares also cannot preserve meaningful relation of those.

Yao garbled circuits operate however on the initial private value, actually on the bits of that value. So the design of the floating point values don't affect Yao Circuit Evaluation protocol (YCE) that much. In general, there are two main factors that impact YCE and its performance: the size of the boolean circuit that represents a function and the size of parties' inputs. Note that if we allow a circuit to be evaluated in parallel then the depth of the circuit plays more important role than actual number of gates.

3 The Method

In this section we present the necessary tools and a brief proof of security of our construction of Yao circut generation/evaluation.

3.1 Array encryption

Usually encryption schemes tend to operate on a single message at a time, so encryption enc and decryption dec operations are defined on the following domains:

$$\mathsf{enc} : \mathcal{K} \times \mathcal{M} \longrightarrow \mathcal{C} \ ,$$
$$\mathsf{dec} : \mathcal{K} \times \mathcal{C} \longrightarrow \mathcal{M} \ ,$$

where $\mathcal{K}$, $\mathcal{M}$ and $\mathcal{C}$ are respectively the key, message and ciphertext spaces. To simplify matters in this paper, we need a modified definition of the encryption scheme, as we need to encrypt tables and arrays so a single cell can be revealed by revealing the keys. Assume, that we have the following setup of four element array arranged in a table, see Figure 1.

	k_x^0	k_x^1
k_y^0	m_{00}	m_{01}
k_y^1	m_{10}	m_{11}

Figure 1: Array encryption table

We have four key values and four messages arranged in the table. Now, we need an array encryption scheme to use two different keys to perform encryption (decryption) operation on each message. To accommodate this fact we modify the domains of the encryption scheme:

$$\mathsf{enc} : \mathcal{K} \times \mathcal{K} \times \mathcal{M} \longrightarrow \mathcal{C} \ ,$$
$$\mathsf{dec} : \mathcal{K} \times \mathcal{K} \times \mathcal{C} \longrightarrow \mathcal{M} \ .$$

The notion $\mathsf{enc}_{k_x,k_y}(m) = c$ means that the message m is encrypted using the keys k_x and k_y to produce the ciphertext c. For the decryption operation we use the same notation with keys written as indexes: $\mathsf{dec}_{k_x,k_y}(c) = m$.
Let us now define a notion for the whole table encryption. Let $\boldsymbol{m} = (m_{00}, m_{01}, m_{10}, m_{11})$, then the following formula describes one of the options how to organize array encryption operation:

$$\mathsf{AE}_{k_x^0,k_x^1,k_y^0,k_y^1}(\boldsymbol{m}) = \begin{pmatrix} \mathsf{enc}_{k_x^0,k_y^0}(m_{00}) & \mathsf{enc}_{k_x^0,k_y^1}(m_{01}) \\ \mathsf{enc}_{k_x^1,k_y^0}(m_{10}) & \mathsf{enc}_{k_x^1,k_y^1}(m_{11}) \end{pmatrix} .$$

Note, that unlike with encryption we want decryption operation to be called on each cell explicitly. More specifically, the security requirement for such array encryption scheme is that an adversary that has two keys, one from each pair (k_x^0, k_x^1) and (k_y^0, k_y^1), should be able to decrypt only one cell. The desired property of the encryption scheme is to be secure under chosen plaintext attacks, so now we define formal games for IND-CPA setting, see Figure 2.

$\mathcal{G}_0^{\mathcal{A}}$

$k_x^0, k_x^1 \leftarrow \mathcal{K} \times \mathcal{K}$
$k_y^0, k_y^1 \leftarrow \mathcal{K} \times \mathcal{K}$
$\boldsymbol{m_0}, \boldsymbol{m_1}, b_x, b_y \leftarrow \mathcal{A}$
if $\boldsymbol{m_0}[b_x][b_y] \neq \boldsymbol{m_1}[b_x][b_y]$ then **return** $\perp$
$c_0 \leftarrow \mathsf{AE}_{k_x^0,k_x^1,k_y^0,k_y^1}(\boldsymbol{m_0})$
$c_1 \leftarrow \mathsf{AE}_{k_x^0,k_x^1,k_y^0,k_y^1}(\boldsymbol{m_1})$
return $\mathcal{A}(c_0, k_x^{b_x}, k_y^{b_y})$

$\mathcal{G}_1^{\mathcal{A}}$

$k_x^0, k_x^1 \leftarrow \mathcal{K} \times \mathcal{K}$
$k_y^0, k_y^1 \leftarrow \mathcal{K} \times \mathcal{K}$
$\boldsymbol{m_0}, \boldsymbol{m_1}, b_x, b_y \leftarrow \mathcal{A}$
if $\boldsymbol{m_0}[b_x][b_y] \neq \boldsymbol{m_1}[b_x][b_y]$ then **return** $\perp$
$c_0 \leftarrow \mathsf{AE}_{k_x^0,k_x^1,k_y^0,k_y^1}(\boldsymbol{m_0})$
$c_1 \leftarrow \mathsf{AE}_{k_x^0,k_x^1,k_y^0,k_y^1}(\boldsymbol{m_1})$
return $\mathcal{A}(c_1, k_x^{b_x}, k_y^{b_y})$

Figure 2: Array encryption IND-CPA games

The advantage of an adversary defined as usual as the following difference:

$$\mathsf{Adv}_{\mathsf{AE}}^{IND-CPA}(\mathcal{A}) = |\mathsf{Pr}[\mathcal{G}_0^{\mathcal{A}} = 1] - \mathsf{Pr}[\mathcal{G}_1^{\mathcal{A}} = 1]|.$$

An encryption scheme is (t, ε)-*array encryption* IND-CPA secure, if for all t-time adversaries $\mathcal{A}$ the advantage $\mathsf{Adv}_{\mathsf{AE}}^{IND-CPA}(\mathcal{A})$ is less or equal to ε.

To build a *one-time pad array encryption scheme*, we need 2ℓ-bit long keys to encrypt four element arrays with ℓ-bit long messages. Namely, we can split each key k into ℓ-bit long blocks and use these blocks sequentially when we need to encrypt message with the key k. To get rid of the restriction on the key length, we use a pseudorandom generator to stretch keys up to the necessary length. If keys are not long enough, we use a pseudorandom generator f to stretch them up to the needed length. The result is given in Protocol 1. For clarity, let $f : \mathcal{K} \to \mathcal{M} \times \mathcal{M}$, particularly $f : \{0,1\}^k \to \{0,1\}^\ell \times \{0,1\}^\ell$, and let $f(x)[0]$ denote the first component of $f(x)$ and $f(x)[1]$ the second component.

Lemma 1. *If a generator used to stretch keys for array encryption scheme is a (t, ε)-pseudorandom generator, the one-time pad array encryption scheme is $(t, 8 \cdot \varepsilon)$-*IND-CPA *indistinguishable.*

Proof. (Sketch) Protocol 2 describes the ideal functionality for array encryption AE°.

Input: $k_x^0, k_x^1, k_y^0, k_y^1$ and message $\boldsymbol{m} = (m_{00}, m_{01}, m_{10}, m_{11})$.
Output: ciphertext $\boldsymbol{c} = (c_{00}, c_{01}, c_{10}, c_{11})$

1. Compute $c_{00} = m_{00} \oplus f(k_x^0)[0] \oplus f(k_y^0)[0]$.
2. Compute $c_{01} = m_{01} \oplus f(k_x^0)[1] \oplus f(k_y^1)[0]$.
3. Compute $c_{10} = m_{10} \oplus f(k_x^1)[0] \oplus f(k_y^0)[1]$.
4. Compute $c_{11} = m_{11} \oplus f(k_x^1)[1] \oplus f(k_y^1)[1]$.

Protocol 1: Array encryption scheme using pseudorandom generator for one-time-pad AE^f

Input: message $\boldsymbol{m} = (m_{00}, m_{01}, m_{10}, m_{11})$
Output: ciphertext $\boldsymbol{c} = (c_{00}, c_{01}, c_{10}, c_{11})$

1. If not provided, generate keys $k_x^0 \leftarrow \{0,1\}^{2\ell}, k_x^1 \leftarrow \{0,1\}^{2\ell}, k_y^0 \leftarrow \{0,1\}^{2\ell}, k_y^1 \leftarrow \{0,1\}^{2\ell}$.
2. Compute $c_{00} = m_{00} \oplus k_x^0[0] \oplus k_y^0[0]$.
3. Compute $c_{01} = m_{01} \oplus k_x^0[1] \oplus k_y^1[0]$.
4. Compute $c_{10} = m_{10} \oplus k_x^1[0] \oplus k_y^0[1]$.
5. Compute $c_{11} = m_{11} \oplus k_x^1[1] \oplus k_y^1[1]$.

Protocol 2: Ideal array encryption scheme AE°

Remember, that the advantage of adversary $\mathcal{A}$ is defined in terms of winning games described in Figure 2. Consider the following intermediate games given on Figure 3.

$\mathcal{G}_2^{\mathcal{A}}$

$k_x^0, k_x^1 \leftarrow \mathcal{M} \times \mathcal{M} \times \mathcal{M} \times \mathcal{M}$
$k_y^0, k_y^1 \leftarrow \mathcal{M} \times \mathcal{M} \times \mathcal{M} \times \mathcal{M}$
$\boldsymbol{m_0}, \boldsymbol{m_1}, b_x, b_y \leftarrow \mathcal{A}$
if $\boldsymbol{m_0}[b_x][b_y] \neq \boldsymbol{m_1}[b_x][b_y]$ then **return** $\perp$
$\boldsymbol{c_0} \leftarrow \mathsf{AE}^\circ_{k_x^0, k_x^1, k_y^0, k_y^1}(\boldsymbol{m_0})$
$\boldsymbol{c_1} \leftarrow \mathsf{AE}^\circ_{k_x^0, k_x^1, k_y^0, k_y^1}(\boldsymbol{m_1})$
return $\mathcal{A}(\boldsymbol{c_0}, k_x^{b_x}, k_y^{b_y})$

$\mathcal{G}_3^{\mathcal{A}}$

$k_x^0, k_x^1 \leftarrow \mathcal{M} \times \mathcal{M} \times \mathcal{M} \times \mathcal{M}$
$k_y^0, k_y^1 \leftarrow \mathcal{M} \times \mathcal{M} \times \mathcal{M} \times \mathcal{M}$
$\boldsymbol{m_0}, \boldsymbol{m_1}, b_x, b_y \leftarrow \mathcal{A}$
if $\boldsymbol{m_0}[b_x][b_y] \neq \boldsymbol{m_1}[b_x][b_y]$ then **return** $\perp$
$\boldsymbol{c_0} \leftarrow \mathsf{AE}^\circ_{k_x^0, k_x^1, k_y^0, k_y^1}(\boldsymbol{m_0})$
$\boldsymbol{c_1} \leftarrow \mathsf{AE}^\circ_{k_x^0, k_x^1, k_y^0, k_y^1}(\boldsymbol{m_1})$
return $\mathcal{A}(\boldsymbol{c_1}, k_x^{b_x}, k_y^{b_y})$

Figure 3: Array encryption IND-CPA games with random number generator

As AE° uses uniformly sampled keys, messages of the tables are xor-ed with random bit-strings, which results in the ciphertext being uniformly sampled from all the possible ciphertexts. Thus, these games are perfectly indistinguishable from each other, which means that there is no

adversary that can distinguish which one of these games it is playing with significant certainty.

Let's investigate what is the computational distance between games $\mathcal{G}_0^{\mathcal{A}}$ and $\mathcal{G}_2^{\mathcal{A}}$. The only significant difference in their description is that encryption is performed in $\mathcal{G}_0^{\mathcal{A}}$ using keys stretched by a pseudorandom generator and for $\mathcal{G}_2^{\mathcal{A}}$ using uniformly random bit-strings. Essentially, the only difference is in how $\boldsymbol{c_0}$ is computed (because adversary does not see $\boldsymbol{c_1}$). Now, in AE^f ciphertext $\boldsymbol{c_0} = F(\boldsymbol{m_0}, f(k_x^0), f(k_x^1), f(k_y^0), f(k_y^1))$, where f is a pseudorandom generator and F is a deterministic function, which actually does the same as AE^{f}, but is written explicitly with the keys as arguments. On the other hand, in AE° the respective ciphertext is computed as $\boldsymbol{c_0^{\circ}} = F(\boldsymbol{m_0}, \bar{k}_x^0, \bar{k}_x^1, \bar{k}_y^0, \bar{k}_y^1)$, where keys are uniformly sampled from the space $\mathcal{M} \times \mathcal{M}$. To derive the computational distance between these games, consider the following series of hybrid games, where the ciphertext $\boldsymbol{c_0}$ is computed in the following fashion:

$$
\begin{aligned}
&\mathcal{G}_{00} : \boldsymbol{c_0} = F(\boldsymbol{m_0}, \bar{k}_x^0, f(k_x^1), f(k_y^0), f(k_y^1)) \ , \\
&\mathcal{G}_{01} : \boldsymbol{c_0} = F(\boldsymbol{m_0}, \bar{k}_x^0, \bar{k}_x^1, f(k_y^0), f(k_y^1)) \ , \\
&\mathcal{G}_{02} : \boldsymbol{c_0} = F(\boldsymbol{m_0}, \bar{k}_x^0, \bar{k}_x^1, \bar{k}_y^0, f(k_y^1)) \ .
\end{aligned}
$$

Now the distance between $\mathcal{G}_0^{\mathcal{A}}$ and $\mathcal{G}_{00}$ is clearly ε. This leads to a contradiction with the function f being a (t, ε)-pseudorandom generator. $\mathcal{B}$ will query the PRG game with k_x^0, then form all necessary input for $\mathcal{A}$ by uniformly sampling three other keys, and return whatever $\mathcal{A}$ returns. This strategy obviously gains the same success, as $\mathcal{A}$ in distinguishing $\mathcal{G}_0^{\mathcal{A}}$ from $\mathcal{G}_{00}$. By a similar argument, the distance between $\mathcal{G}_{00}$ and $\mathcal{G}_{01}$ is ε, the distance between $\mathcal{G}_{01}$ and $\mathcal{G}_{02}$ is ε and the distance between $\mathcal{G}_{02}$ and $\mathcal{G}_2^{\mathcal{A}}$ is ε. Thus, the total distance between $\mathcal{G}_0^{\mathcal{A}}$ and $\mathcal{G}_2^{\mathcal{A}}$ is $4 \cdot \varepsilon$.

In the same manner we show that the distance between $\mathcal{G}_1^{\mathcal{A}}$ and $\mathcal{G}_3^{\mathcal{A}}$ is $4 \cdot \varepsilon$. Thus, the total computational distance between games $\mathcal{G}_0^{\mathcal{A}}$ and $\mathcal{G}_1^{\mathcal{A}}$ is then $8 \cdot \varepsilon$, so the encryption scheme is $(t, 8 \cdot \varepsilon)$-IND-CPA indistinguishable. □

3.2 Yao garbled circuits

Consider the following modified scenario of secure two-party computation. Two parties $\mathcal{P}_1$ and $\mathcal{P}_2$ want to compute $(\varnothing, z) \leftarrow f(x_1, x_2)$, where x_1 and x_2 are respective parties' inputs. An important difference from a general two-party computation case is that only one party is going to receive a meaningful output value, as sender $\mathcal{P}_1$ always receives $\varnothing$ if protocol run completes.

The first general solution to such secure two party computation problem was proposed by A.C.C. Yao in the seminal paper [1]. His idea consists of three main techniques:

- reordering a gate's truth table rows to hide the content of the gate;
- modifying the boolean circuit to operate on encryption scheme keys, not just bit values;
- using oblivious transfer to securely give an evaluation party input to the circuit.

In this paper we use a xor + PRG array encryption scheme AE_f described before. This simplifies the description of Yao garbled circuit construction procedure and is somewhat easier to understand for a reader.

3.3 Yao garbled circuit construction

Flipping values on wires. We start securing boolean circuit evaluation with the following procedure. In the evaluation process, every wire in circuit will hold one bit value. We need those values not to provide any meaningful information to the adversary. To do that we will randomly modify the meaning of this value. The procedure to do that follows: for every wire, with probability one half, we "flip" its values. Flipped values are then detached from the actual meaning of the value on the wire, because a wire which must hold value (for instance 1) can contain either 1 or 0 after being flipped.
If we do not modify anything else in the circuit, obviously the evaluation will not compute the intended function correctly. Thus, we must modify the content of the truth-tables of gates accordingly. Suppose we have an and gate with its ordinary truth-table. This gate has two input wires and one output wire. Suppose then that we have flipped the value of the first input of the gate (wire x in the Table 1), then 0 on that wire actually means 1. Suppose the flipped wire x is denoted by $\bar{x}$, then we modify the table in the following way.

$\bar{x}$	y	$x \wedge y$
0	0	0
0	1	1
1	0	0
1	1	0

Table 1: and gate truth table with flipped x wire

Note, that we have changed the contents of the result column of that table. If the values on wires y or $x \wedge y$ have also been flipped, we would need to also take that into account when computing the table.

Extending values on wires. The procedure for this step of securing circuit evaluation is the following. For every wire w of the circuit, we generate two random keys for the array encryption scheme. We just create two random keys for each wire. Then we use the gates' reorganized truth-tables from the previous step, and substitute bit values in those with corresponding wire keys. Let k_x^0, k_x^1, k_y^0, k_y^1 be keys generated for wires x and y respectively. Additionally, let k_z^0, k_z^1 be keys generated for gate's output wire z.

Also, let f be a pseudorandom generator that we will feed to the array encryption scheme to stretch keys. Then we put those keys into the truth-table of the gate and encrypt values of the output column (see Table 2). As in the previous example, we have wire x flipped, and denote that by using $\bar{x}$ in the gate's truth-table. Note, that the output wire key is concatenated with

$\bar{x}$	y	$x \wedge y$
k_x^0	k_y^0	$\mathsf{enc}_{k_x^0,k_y^0}(k_z^0 \| \| 0)$
k_x^0	k_y^1	$\mathsf{enc}_{k_x^0,k_y^1}(k_z^1 \| \| 1)$
k_x^1	k_y^0	$\mathsf{enc}_{k_x^1,k_y^0}(k_z^0 \| \| 0)$
k_x^1	k_y^1	$\mathsf{enc}_{k_x^1,k_y^1}(k_z^0 \| \| 0)$

Table 2: and gate encrypted truth table

a bit value, so domains for array encryption scheme are then: $\mathcal{K} = \{0,1\}^{128}$, $\mathcal{M} = \{0,1\}^{129}$, $\mathcal{C} = \{0,1\}^{129}$; for AES with 128-bit keys as pseudorandom generator.

The output column of this table is filled with pseudorandom data. Garbled circuit generation is finished now and description of the circuit will consist of:

- the description of wires, from which gate it goes to which;
- the description of gates: encrypted flipped truth table result column.

Note that in order to reverse the output of the circuit, which will be a key value, the circuit generator, also known as generating party, must remember the corresponding key to bit value transitions and if the output wire was flipped. This means that if the circuit generator has flipped values on the output wires of the circuit, it must remember which wire it has flipped, because then the value produced by circuit evaluation could be a flipped one.

Transferring keys to the evaluator. This circuit cannot be evaluated without knowing the keys corresponding to a values for input wires. The party that generated the circuit can send keys corresponding to its input with the circuit description. They do not reveal bits of input, so it is still private. On the other hand, parties need to engage in an oblivious transfer protocol to get keys that correspond to circuit evaluating party's input to evaluator.
The setting for Yao garbled circuit protocol is such, that generating party has two keys and the evaluating party has a bit value for each input wire. During oblivious transfer for every bit of the evaluator's input value it will learn a corresponding key value and the generator will learn nothing.
To evaluate a garbled circuit, the evaluator receives its description and keys corresponding to the input of the generating party. The evaluator then obtains the keys corresponding to its own input using an oblivious transfer protocol. Then the evaluator is able to evaluate the circuit by decrypting all its gates one by one. If there is a gate where all input wires contain known key values, the evaluator can decrypt the truth table. As every row of the truth table is encrypted with different key pairs it will be able to decrypt only one row. The decrypted value specifies keys for the outgoing values of this gate. Proceed with such gate by gate evaluation until no non-evaluated gates are left. Collect circuit output value from the circuit's output wires. For every output wire w these values will be in the form of $out_w = key \mid\mid b$, where key is one of the keys generated for output wires and b is a bit value of the output. Note, that keys contained

on an input or intermediate wires are used in the circuit evaluation. However, we do not use values of output wires, because nothing is encrypted with those. So the generator of the circuit can put any bit-string of corresponding length there. The array encryption scheme works fine on any bit-strings and can actually be used to encrypt messages of arbitrary length, so it does not limit the information that can be put on an output wire. Also note, that output wires can be either flipped or not. The evaluator's ability to learn the actual output depends of how the setting is fixed. If the values on output wires were possibly flipped, it will not know the output. Otherwise, the evaluator will know the actual output.

3.4 Oblivious transfer in Sharemind

Oblivious transfer in Sharemind can be implemented in the following way. Let's refer the miners of Sharemind as $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ respectively. Additionally, we use the envelope notation $[\![x]\!]$ to denote operations performed on a shared value x. A result of an operation on shared values is also a shared value, so, for example, notion $[\![z]\!] = [\![x]\!] + [\![y]\!]$ means that parties compute a shared value for z as a sum of their shared values for x and y. Suppose now that $\mathcal{P}_1$ has two values x_0 and x_1 and $\mathcal{P}_2$ knows bit b. We want to perform an oblivious transfer so $\mathcal{P}_2$ learns x_b, but not x_{1-b} and no miner learns b. To do so, $\mathcal{P}_1$ shares x_0 and x_1 between miners and $\mathcal{P}_2$ shares b. Then miners compute message m.

$$[\![m]\!] = [\![x_0]\!] \cdot ([\![1]\!] - [\![b]\!]) + [\![x_1]\!] \cdot [\![b]\!] = \begin{cases} [\![x_0]\!] & \text{if } b = 0 \\ [\![x_1]\!] & \text{if } b = 1 \end{cases}$$

When this is done all miners communicate their shares of m to $\mathcal{P}_2$ which then learns x_b.

3.5 Summary

Having all the tools described above, we can implement Yao circuits protocols for Sharemind. Our implementation follows exactly the procedure of generating a Yao garbled circuit described above. This construction is information-theoretically secure against passive adversaries [2]. We provide a proof-of-concept implementation of YCE and its integration into the Sharemind environment.

4 Results

We have successfully created a Python implementation of the Yao Garbled Circuits protocol. The implementation is compatible with version 2.6.2 (and several other versions) of Python.

4.1 YCE implementation

Our implementation of the YCE uses AES key stretching procedure to generate randomness. Other implementation details correspond to a method of generating/evaluating garbled circuits

described earlier in this paper.

Our implementation is able to parse a specific format for describing boolean circuits (this format is described in the following sections) and is verified to work on circuits up to few thousands of logical gates.

4.2 Integration with Sharemind

Integration with Sharemind is done via a typical C++/Python integration. Sharemind code imports the *Python.h* header and is linked with a Python dynamic libraries. This allows Sharemind to create a Python interpreter and execute Python scripts programmatically. Communication between miners (which includes oblivious transfer of the input keys) is done on the Sharemind's part of the solution and uses general patterns of inter-miners' communication.

Oblivious transfer is done in Sharemind and its implementation corresponds exactly to a method described earlier in this paper.

4.3 The circuit description format

A circuit consists of several boolean expressions. Actually, we think of a circuit as a set of boolean expressions one per each output.

Consider the following example of full-adder circuit description:

$$
\begin{aligned}
&g1: \ xor\ a\ b\\
&g2: \ xor\ g1\ c\\
&g3: \ and\ a\ b\\
&g4: \ and\ c\ g1\\
&g5: \ or\ g3\ g4\\
&add2\ g2\ g5
\end{aligned}
$$

The last line contains the circuit's name $add2$ and describes two circuit's outputs. Now we don't want to describe each output as one big expression as it would be inefficient. So we introduce a concept of a "named gate/expression".

Named expression is a boolean expression that has a name assigned to it. In the example above - $g1$, $g2$ etc are named gates. A named gate is described in the following way: *gatename* : *expression*. (note the colon), where thhe name of the gate can contain only alphanumeric characters. An expression may be complex (like: $and\ u28\ and\ a27\ not\ b27$) and not just a single operation.

The whole point of having a named gate is to reuse it. As you can see, $g1$ is used twice, in the $g2$ and $g4$ definitions.

Now, all arguments for any expression, that are not a named expressions themselves (like a, b, c in the example above) are assumed to be circuit inputs. Remember, that those are single

bits, so if you need an int input a, a comfortable way to represent that will be bits: $a0$, $a1$, ..., $a31$.

Possible operations for expressions are: *and*, *or*, *xor*, *eq*, *nand* and *unary not*.

When a circuit is parsed, one gate will be created for every operation. Also, to ensure that inputs can be used more than once in the circuit, an identity gate will be created for every input value.

We must note that due to constraints of generating randomness using AES key stretching procedure, our proof-of-concept implementation can lead to undefined behavior, when encounter a gate with more than 4 output wires.

There are example circuits created for the following operations:

1. full adder;
2. sum of two 4 bit integers;
3. greater-than or equal for two 32 bit integers;
4. sum of three 32 bit integers;

There are additional constraints for circuits created to be evaluated from Sharemind. Currently our implementation effortlessly supports binary operations on a shared values. Consider an operation on the shared values $[\![A]\!] * [\![B]\!] = [\![C]\!]$. As boolean circuit needs to assign a name for every single input value, out implementation follows a convention:

- $a = [\![A]\!]_1$
- $b = [\![A]\!]_2$
- $c = [\![A]\!]_3$
- $d = [\![B]\!]_1$
- $e = [\![B]\!]_2$
- $f = [\![B]\!]_3$
- $x = [\![C]\!]_1$
- $y = [\![A]\!] * [\![B]\!] - (x + z)$
- $z = [\![C]\!]_3$

Note that, y looks differently from every other variable. It is considered to be an output of the circuit and must hold a share of the operation's result. As shares of miners 1 and 3 are known to them beforehand y gets the view above.

Following this convention will ensure that one can implement all kinds of binary functions for Sharemind without implementing them in low-level C.

We have implemented two sample circuits for addition of 32 bit integers (represented as shares) and comparison of a value with 0, where value is also shared between miners.

4.4 Performance details

Tests were executed on a 32 bit Windows 7 Professional machine with Intel Core i5 CPU (2.67 GHz) and 4.00 GB of memory. This test measured only our Python YCE implementation, with no integration with Sharemind. Oblivious transfer protocols were simulated with a negligible performance overhead, so actual performance within the Sharemind environment can slightly decrease due to that. Tests were performed on two sample functions: *shared sum* - a sum of 2 32 bit shared integers and *shared gt* - a greater than function on 2 shared integers. The result is shown in the table 3.

Function	Gates	Generation (s)	Evaluation (s)	Total time (s)
shared sum	1419	1.138	0.345	1.483
shared gt	1329	1.017	0.319	1.336

Table 3: YCE performance results

The column *Gates* shows the number of gates the function's circuit consists of, *Generation*, *Evaluation* show time taken by the generation and the evaluation of the circuit. Further profiling of the implementation showed that the average time to garble a circuit was much less than the one presented above (about 0.67 s). However, to integrate with Sharemind our implementation needs to post-process the circuit and the keys and that part also takes quite some time. Additionally, we want to note, that most of the time were spent initializing AES for the randomness.

Overall results show that it is definitely a proof-of-concept implementation, which performance can be greatly improved.

References

[1] Andrew Chi Chih Yao. *Protocols for secure computations (extended abstract)*. In FOCS, pp. 60164, 1982.

[2] Oleg Šelajev. *The Use of Circuit Evaluation Techniques for Secure Computation*. Tartu University, Master's thesis, 2011.