CYBERNETICA
Institute of Information Security

A general mechanism for implementing secure operations on secret shared data

Sander Siim, Dan Bogdanov

T-4-21 / 2014

Sander Siim[1,2], Dan Bogdanov[1].
[1] Cybernetica, Institute of Information Security,
[2] University of Tartu, Institute of Computer Science,

This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. Distribution Statement A (Approved for Public Release, Distribution Unlimited).

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

Mailing address:
Cybernetica AS
Mäealuse 2/1
12618 Tallinn
Estonia

A general mechanism for implementing secure operations on secret shared data

Sander Siim, Dan Bogdanov

18th March 2014

Abstract

This report describes the prototype implementation of a generic mechanism for creating secure multiparty computation (SMC) protocols that work on secret-shared data. The standard method for designing efficient protocols for secure computation based on secret sharing includes the clever use of algebraic properties to enable fast implementations on current hardware. However, these kinds of protocols are often less flexible with regard to bit level access. On the other hand, Yao-style garbled boolean circuits are very flexible, but can be less efficient. The goal of this work is to balance the low computational complexity of secret sharing with the high flexibility of garbled boolean circuits. We will build a hybrid protocol that enables garbled boolean circuit evaluation on bitwise secret shared data. The report also describes an implementation of this capability within the Sharemind 3 runtime and its initial benchmarking results.

Contents

1 Design of a hybrid protocol

1.1 Foundations

1.1.1 Protocols based on secret sharing

Secret sharing is a well-known cryptographic method for distributing a secret amongst a group of parties [Sha79]. The secret is divided into *shares* and each party receives a share of the secret, which appears random to the receiving party. The goal for any secret sharing scheme is that the secret can only be reconstructed by combining a sufficiently large subset of the shares. For a *k-out-of-n secret sharing scheme*, the secrets are divided into n shares and knowing any $k-1$ shares does not reveal the original secret. Secret sharing schemes can be used in secure multiparty computation to build protocols that guarantee data privacy by performing computations on secret-shared data [BOGW88, CCD88].

SMC protocols based on secret sharing require multiple computing parties since the data is distributed. To ensure data privacy, the computing parties must not learn the shares of the other parties, otherwise they could reconstruct the secret shared data. For that end, the protocol performs distributed secure computations on the shares. The general construction for a SMC protocol using secret sharing is the following [Bog13]:

1. An input party $\mathcal{IP}$ divides its data into n shares
2. $\mathcal{IP}$ sends a share to each computing party $\mathcal{CP}_i$
3. The computing parties $\mathcal{CP}_1,\dots,\mathcal{CP}_n$ perform secure computations on the shares
4. The computing parties send the output shares to a result party $\mathcal{RP}$
5. $\mathcal{RP}$ receives the shares and reconstructs the actual output

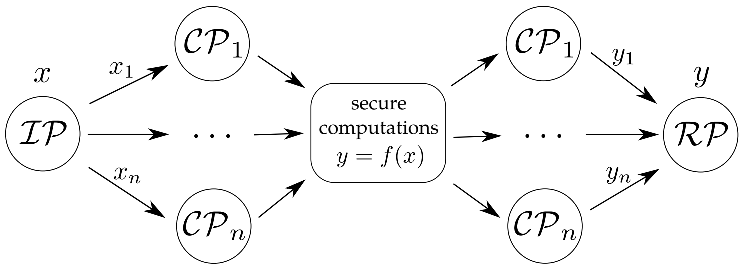

Figure 1: General construction of secret sharing protocols

Such a protocol calculates a specific function f, which is determined by the computations performed in step 3. To build a protocol which calculates a different function g, different secure computations must be specified. Constructing such protocols is not always straightforward and requires careful manipulation of the shares to ensure data privacy. For example, constructing a division protocol for secret-shared integers is quite a challenging task [BNTW12].

1.1.2 Protocols based on garbled boolean circuits

Yao's garbled boolean circuit protocol is a widely used and provably secure solution to the two-party secure computation problem [LP04]. It allows secure function evaluation for two parties, that is to compute a function on their inputs without revealing one's inputs to the other.

In Yao's protocol, the computed function f is presented as a boolean circuit. A boolean circuit is a set of gates and wires. Wires connect gates and transfer bit values between them. Each gate has a number of input wires and output wires. If a gate has n input wires and m output wires, then the gate calculates a function $f : \{0,1\}^n \to \{0,1\}^m$. In practice, mostly 2-to-1 and 1-to-1 gates are used. The gate calculates the function by receiving the input bits from its input wires and storing the output to its output wires.

To illustrate Yao's protocol, let us consider the situation where parties $\mathcal{A}$ and $\mathcal{B}$ want to compute some function f on their respective secret inputs a and b. Both parties want to learn the result $f(a,b)$, but neither wants to disclose his input to the other party. Let f be presented as a boolean circuit. One party will take the role of the *garbler* and the other the *evaluator*. Let $\mathcal{A}$ be the garbler and $\mathcal{B}$ the evaluator.

The essence of Yao's protocol is that the garbler uses symmetric encryption to encrypt the truth table values of the circuit's gates. This process is called *garbling*. For each wire the garbler generates two tokens X^0, X^1 which represent the bit values of the wire - one token for bit value 0 and one for bit value 1. However, the token itself does not reveal to which bit it maps to. The circuit can then be evaluated only if the secret tokens of the input wires that correspond to the given input bits are known.

The garbled truth tables are then sent to the evaluator along with the input tokens of the garbler's input. Oblivious transfer is used to transport the input tokens corresponding to evaluator's input to the evaluator, which guarantees that the garbler does not find out the evaluator's input. The evaluator can then evaluate the circuit gate-by-gate without actually learning the garbler's input, since during evaluation, the tokens that represent bit values are passed along wires, not the bit values themselves.

Finally, as the result of the evaluation the evaluator receives $f(a,b)$. In the semi-honest model, the garbler is also guaranteed to receive the output, as the evaluator simply sends it to him. Additionally, the output wires' values can also be encrypted by the garbler. In that case, the evaluator sends the result of the evaluation back to the garbler, who can decrypt it to receive the actual output. Again, in the semi-honest model, the garbler is guaranteed to send the actual output back to the evaluator.

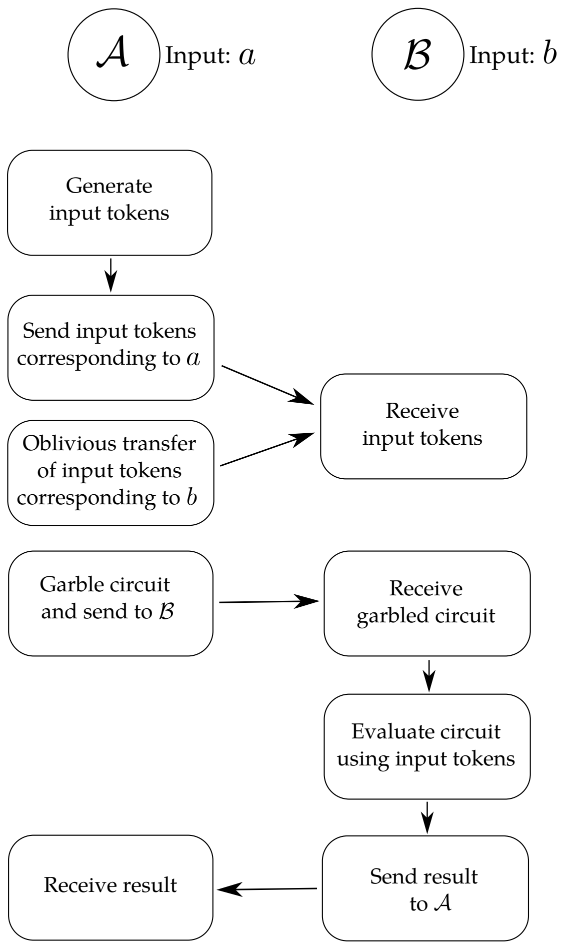
$\mathcal{A}$
Input: a
$\mathcal{B}$
Input: b
Generate input tokens
Send input tokens corresponding to a
Oblivious transfer of input tokens corresponding to b
Receive input tokens
Garble circuit and send to $\mathcal{B}$
Receive garbled circuit
Evaluate circuit using input tokens
Send result to $\mathcal{A}$
Receive result

Figure 2: General construction of Yao garbled circuit protocols

Using Yao's protocol, two parties can compute virtually any function in a privacy-preserving manner, provided that the boolean circuit representation of the required function is known. Building efficient implementations of Yao's protocol has been a significant challenge in the past, but over the past years, many succesful implementations have emerged [MNPS04, HKS$^+$10, HEKM11, KSS12].

1.2 The hybrid protocol

The goal of this document is to describe a general SMC protocol which allows secure evaluation of any function on bitwise secret-shared data, meaning that every bit of the data is shared independently. We will call this the *hybrid protocol*, since it combines the efficiency of secret sharing with the robustness of Yao's garbled circuit approach. Note that in the scope of this document, we are interested only in security in the *semi-honest model*, where computing parties are *honest* in the sense that they strictly follow the protocol, but *curious*, meaning that they will try to use all data available to them to learn as much as they can.

We will give the description of the protocol in the Sharemind multiparty computation setting. Sharemind uses three computing parties, which run secure protocols on secret-shared data [Bog13]. Sharemind operates on a 3-out-of-3 secret sharing scheme, which means that in order to reconstruct data which is shared between the computing parties, all three shares are needed. We will denote the computing parties as $\mathcal{CP}_1$, $\mathcal{CP}_2$ and $\mathcal{CP}_3$. In the hybrid protocol, $\mathcal{CP}_1$ and $\mathcal{CP}_2$ will take the roles of the garbler and evaluator respectively from the Yao garbled circuit protocol. Since the hybrid protocol operates on secret shared data, we will use $\overline{[\![b]\!]} = [\![b_1, b_2, \ldots, b_n]\!]$ to denote a bit vector with length n which is shared between the miners, where $[\![b_i]\!]$ represents the i-th shared bit from the vector. Note that each bit b_i is shared individually.

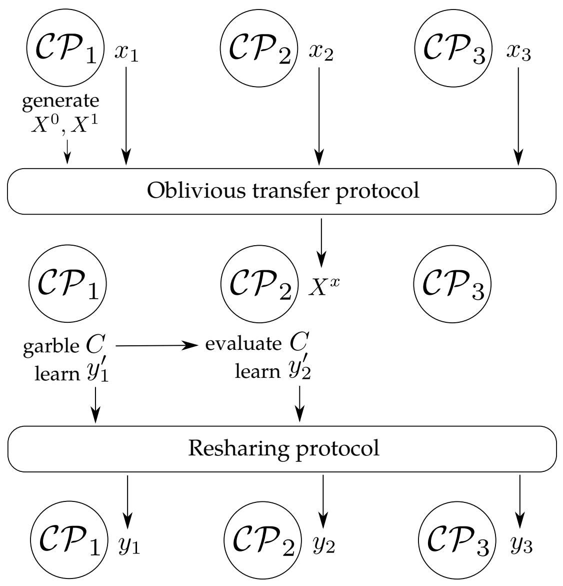

Figure 3: Hybrid protocol overview

Now, suppose we have a bit vector $\overline{[\![x]\!]} = [\![x_1, \ldots, x_n]\!]$ shared between the computing parties and we want to compute the result $f(\overline{[\![x]\!]}) = \overline{[\![y]\!]} = [\![y_1, \ldots, y_m]\!]$, where f is a function $f : \{0,1\}^n \to \{0,1\}^m$. Let us assume that all computing parties have access to a boolean circuit C which calculates the function f. Since the garbling process produces a pair of tokens for each wire, we will use X_j^b to denote the token of the j-th wire corresponding to bit $b \in \{0,1\}$. We say X_j^b has the *semantics* of b. The high-level description of the hybrid protocol is given as

Algorithm 1: Hybrid protocol

```
Input: Shared bit vector $\overline{[\![x]\!]} = [\![x_1, \dots, x_n]\!]$
Boolean circuit $C$ for the function $f : \{0,1\}^n \to \{0,1\}^m$
Output: Shared bit vector $\overline{[\![y]\!]} = [\![y_1, \dots, y_m]\!]$ such that $\overline{[\![y]\!]} = f(\overline{[\![x]\!]})$
foreach input wire $i \in \{1, \dots, n\}$ do
    $\mathcal{CP}_1$ generates token pair $(X_i^0, X_i^1)$
The computing parties initiate an oblivious transfer protocol which results in
$\mathcal{CP}_2$ receiving $\{X_1^{x_1}, \dots, X_n^{x_n}\}$ (the input tokens corresponding to the actual
input bits)
$\mathcal{CP}_1$ garbles circuit $C$ and sends the garbled truth tables to $\mathcal{CP}_2$
$\mathcal{CP}_2$ evaluates garbled $C$ using input tokens $\{X_1^{x_1}, \dots, X_n^{x_n}\}$ and receives
output wires' tokens $\left\{X_{o_1}^{y_1}, \dots, X_{o_m}^{y_m}\right\}$
The computing parties produce their output shares and reshare the output to
receive $\overline{[\![y]\!]}$
return $\overline{[\![y]\!]}$
```

Algorithm 1.

Note, that unlike the original Yao's garbled circuit protocol where input comes from both the garbler and evaluator, here the input is secretly shared between all miners as well as the output. For oblivious transfer between computing parties, the protocol makes use of Sharemind's secure multiplication and addition protocols on secret-shared integers to perform an oblivious choice.

Since we do not want any of the computing parties to actually learn the output of the calculation, the output wires' values are also encrypted by the garbler, but the evaluator does not send the result of the evaluation back to the garbler. Instead, the computing parties will calculate random shares of the output using a *perfectly secure resharing protocol.* This will guarantee the *perfect security* of our protocol [Bog13].

1.3 Parts of the hybrid protocol

The next sections will cover different parts of the hybrid protocol in detail. We will also give a proof of the *perfect security* of our protocol following the blueprint described in [Bog13], showing that the hybrid protocol is *perfectly simulatable* and is followed by a *perfectly secure output resharing protocol.*

1.3.1 Oblivious transfer

Let us first consider the oblivious transfer of the input tokens. The input to the protocol is $[\![x_1, \dots, x_n]\!]$ and the garbler $\mathcal{CP}_1$ has generated corresponding input tokens $\left\{X_1^0, \dots, X_n^0, X_1^1, \dots, X_n^1\right\}$. We now need an oblivious transfer protocol from $\mathcal{CP}_1$ to $\mathcal{CP}_2$ which satisfies the following conditions:

1. As the result, $\mathcal{CP}_2$ should learn $\{X_1^{x_1}, \dots, X_n^{x_n}\}$ and nothing else

Algorithm 2: Oblivious transfer of input tokens

```
Input: $\mathcal{CP}_1$ holds the input tokens $\{X_1^0, \dots, X_n^0, X_1^1, \dots, X_n^1\}$
The input bit vector $\llbracket x \rrbracket = \llbracket x_1, \dots, x_n \rrbracket$ is shared between all parties
Output: $\mathcal{CP}_2$ receives input tokens $\{X_1^{x_1}, \dots, X_n^{x_n}\}$
$\llbracket X^0 \rrbracket = \llbracket X_1^0, \dots, X_n^0 \rrbracket$ and $\llbracket X^1 \rrbracket = \llbracket X_1^1, \dots, X_n^1 \rrbracket$ are instantiated as shared values, with $\mathcal{CP}_1$ taking as his shares the actual tokens and $\mathcal{CP}_2$, $\mathcal{CP}_3$ taking zero-shares
$\llbracket \hat{x} \rrbracket = \llbracket 1 \rrbracket - \llbracket x \rrbracket$
$\llbracket Y' \rrbracket = \llbracket X^0 \rrbracket \cdot \llbracket \hat{x} \rrbracket$
$\llbracket Y'' \rrbracket = \llbracket X^1 \rrbracket \cdot \llbracket x \rrbracket$
$\llbracket Y \rrbracket = \llbracket Y' \rrbracket + \llbracket Y'' \rrbracket$
$\llbracket Y \rrbracket$ is declassified to $\mathcal{CP}_2$ as $\mathcal{CP}_1$ and $\mathcal{CP}_3$ send their shares of $\llbracket Y \rrbracket$ to $\mathcal{CP}_2$
$\mathcal{CP}_2$ combines the shares of $\llbracket Y \rrbracket$ to get $\{X_1^{x_1}, \dots, X_n^{x_n}\}$
return $\{X_1^{x_1}, \dots, X_n^{x_n}\}$
```

2. $\mathcal{CP}_1$ must not learn $\{X_1^{x_1}, \dots, X_n^{x_n}\}$, i.e, which tokens were transferred to $\mathcal{CP}_2$

3. No computing party can learn the input $\{x_1, \dots, x_n\}$

It can be seen easily that if we had an oblivious choice protocol which follows conditions 2., 3. and outputs $\llbracket X_1^{x_1}, \dots, X_n^{x_n} \rrbracket$ shared between the computing parties, then satisfying condition 1. is trivial, since we can extend the oblivious choice by simply sending all result shares from other computing parties to $\mathcal{CP}_2$. Performing an oblivious choice on secret-shared data however can be easily implemented using secure multiplication and addition protocols. The resulting oblivious transfer protocol is described in Algorithm 2.

On lines 2-5, Sharemind's secure multiplication and addition protocols are used to perform an oblivious choice. The calculations can be summarized as $\llbracket Y \rrbracket = \llbracket X^0 \rrbracket \cdot (\llbracket 1 \rrbracket - \llbracket x \rrbracket) + \llbracket X^1 \rrbracket \cdot \llbracket x \rrbracket$, but for clarity, separate protocol calls are written on different lines. As the result, $\llbracket Y \rrbracket$ contains the necessary tokens which need to be transferred to $\mathcal{CP}_2$. Then on line 6, the shares of $\llbracket Y \rrbracket$ are sent to $\mathcal{CP}_2$ who can combine them to receive the actual input tokens. Note that the tokens X^i are bit strings of length k. Although Sharemind's multiplication and addition are defined on elements of a ring $\mathbb{Z}_{2^n}$, we can decode the tokens as an array of $\mathbb{Z}_{2^n}$ elements, and extend $\llbracket x \rrbracket$ and $\llbracket \hat{x} \rrbracket$ to match the extended length of the tokens.

We will prove in Section 1.5 that the oblivious transfer protocol described in Algorithm 2 is perfectly simulatable.

1.3.2 Garbling

After the oblivious transfer of the input tokens to the evaluator, the hybrid protocol is very similar to a standard Yao garbled circuit protocol and can be implemented using various garbling schemes and circuit formats.

Our implementation uses the garbling scheme GaXR presented by Bellare et al. [BHKR13], which is based on modeling fixed-key AES as a random permutation. Bellare et al.'s garbling scheme is one of the most efficient garbling schemes to date and takes full advantage of hardware with AES-NI support. Also, the chosen scheme is compatible with the well-known *free-XOR* [KS08] and *garbled row reduction* [PSSW09] optimizations. For encryption, we chose A4 over other alternatives presented in the paper since it helps reduce network communication, which we expected to be a performance bottleneck. Note that this garbling scheme is defined for a specific circuit construction. The circuit must consist of only 2-to-1 gates with arbitrary fan-out and functionality. Each non-input wire must be an outgoing wire of a gate. Circuit's output wires cannot be input wires or inputs to gates. All input wires and output wires are unique and no wire can twice feed a gate.

Following the notation of [BHKR13] each circuit C can be described as a tuple (n, m, q, A, B, G), where n is the number of input wires, m the number of output wires and q the number of gates in C. Then Inputs $= [1, \ldots, n]$, Wires $= [1, \ldots, n+q]$, OutputWires $= [n+q-m+1, \ldots, n+q]$ and Gates $= [n+1, \ldots, n+q]$. A and B are functions Gates $\to$ Wires$\backslash$InputWires which respectively identify the first and second input wire of any gate. G is a function Gates $\times \{0,1\}^2 \to \{0,1\}$ which determines the functionality of each gate. We have presented detailed algorithms of the whole protocol for each computing party in Figure 4.

For each input wire $i \in$ Inputs, the garbler $\mathcal{CP}_1$ generates a token pair (X_i^0, X_i^1) with X_i^0 and X_i^1 having the semantics of 0 and 1 respectively. We will call the last bit of a wire token it's *value bit*, since the evaluator uses it to choose which row in the garbled truth table to decrypt. However, to hide the true semantics of the tokens from the evaluator, each token's value bit is masked with a random bit $p_i \xleftarrow{\$} \{0,1\}$, which is called a *permutation bit*. We will denote a token X_i^b with a value bit p_i as $X_i^b|p_i$. Finally, this results in tokens $(X_i^0|p_i, X_i^1|\overline{p_i})$ for wire i, with value bits p_i and $\overline{p_i}$ respectively.

Since we are using the free-XOR technique, the input tokens are generated using a global random token $R \xleftarrow{\$} \{0,1\}^{k-1} \parallel 1$, where the last bit of R is always 1. This guarantees that tokens with different semantics also have different value bits, as X_i^1 is generated as $X_i^1 \leftarrow X_i^0 \oplus R$.

After the input tokens have been generated, each computing party participates in the oblivious transfer of the input tokens to $\mathcal{CP}_2$. We use OT to denote the call to the oblivious transfer protocol. Each party inputs their shares of $[\![x]\!]$ to the OT protocol and $\mathcal{CP}_1$ also inputs the generated input tokens. Note that the OT protocol is run simultaneously on all three computing parties.

After completing the oblivious transfer, $\mathcal{CP}_1$ starts garbling the circuit and produces the encrypted truth tables of all gates, which are saved in P. For each gate g, the encrypted output corresponding to input tokens with value bits $i, j \in \{0,1\}$ is stored in $P[g, i, j]$. However, XOR-gates are not encrypted, but

Algorithm 3: Hybrid protocol view of $\mathcal{CP}_1$

```
Input: Input shares
       $x_{*1} = [x_{11}, \dots, x_{n1}]$ and circuit
       $C_f = (n, m, q, A, B, G)$
Output: Shares $[y_{11}, \dots, y_{m1}]$ of $[\![y]\!]$
        such that $[\![y]\!] = f([\![x]\!])$
$R \xleftarrow{\$} \{0,1\}^{k-1} \parallel 1$
for $i \leftarrow 1$ to $n$ do
    $p_i \xleftarrow{\$} \{0,1\}$
    $X_i^0 \xleftarrow{\$} \{0,1\}^{k-1} \parallel p_i, \quad X_i^1 \leftarrow X_i^0 \oplus R$
$\mathtt{OT}\ ([X_1^0, \dots, X_n^0], [X_1^1, \dots, X_n^1], x_{*1})$
for $g \leftarrow n+1$ to $n+q$ do
    $a \leftarrow A(g), \quad b \leftarrow B(g)$
    if $G_g = XOR$ then
        $X_g^0 \leftarrow X_a^0 \oplus X_b^0, \quad X_g^1 \leftarrow X_g^0 \oplus R$
    else
        for $i \leftarrow 0$ to $1, j \leftarrow 0$ to $1$ do
            $u \leftarrow i \oplus \mathtt{lsb}(X_a^0)$
            $v \leftarrow j \oplus \mathtt{lsb}(X_b^0)$
            $r \leftarrow G_g(u, v)$
            if $i = 0$ and $j = 0$ then
                $X_g^r \leftarrow$
                $\mathtt{Enc}(X_a^u, X_b^v, g, 0^k)$
                $X_g^{r-1} \leftarrow X_g^r \oplus R$
            else
                $P[g, i, j] \leftarrow$
                $\mathtt{Enc}(X_a^u, X_b^v, g, X_g^r)$
Send $P$ to $\mathcal{CP}_2$
for $i \leftarrow 1$ to $m$ do
    $y'_{i1} \leftarrow \mathtt{lsb}(X^0_{n+q-m+i})$
$[y_{11}, \dots, y_{m1}] \leftarrow$
$\mathtt{Reshare}([y'_{11}, \dots, y'_{m1}])$
return $[y_{11}, \dots, y_{m1}]$
```

Algorithm 4: Hybrid protocol view of $\mathcal{CP}_2$

```
Input: Input shares
       $x_{*2} = [x_{12}, \dots, x_{n2}]$ and circuit
       $C_f = (n, m, q, A, B, G)$
Output: Shares $[y_{12}, \dots, y_{m2}]$ of $[\![y]\!]$
        such that $[\![y]\!] = f([\![x]\!])$
$[X_1, \dots, X_n] \leftarrow \mathtt{OT}\ (0^{k \cdot n}, 0^{k \cdot n}, x_{*2})$
Receive $P$ from $\mathcal{CP}_1$
for $g \leftarrow n+1$ to $n+q$ do
    $a \leftarrow A(g), \quad b \leftarrow B(g)$
    $i \leftarrow \mathtt{lsb}(X_a), \quad j \leftarrow \mathtt{lsb}(X_b)$
    if $G_g = XOR$ then
        $X_g \leftarrow X_a \oplus X_b$
    else if $i = 0$ and $j = 0$ then
        $X_g \leftarrow \mathtt{Enc}(X_a, X_b, g, 0^k)$
    else
        $X_g \leftarrow \mathtt{Dec}(X_a, X_b, g, P[g, i, j])$
for $i \leftarrow 1$ to $m$ do
    $y'_{i2} \leftarrow \mathtt{lsb}(X_{n+q-m+i})$
$[y_{12}, \dots, y_{m2}] \leftarrow$
$\mathtt{Reshare}([y'_{12}, \dots, y'_{m2}])$
return $[y_{12}, \dots, y_{m2}]$
```

Algorithm 5: Hybrid protocol view of $\mathcal{CP}_3$

```
Input: Input shares
       $x_{*3} = [x_{13}, \dots, x_{n3}]$
Output: Shares $[y_{13}, \dots, y_{m3}]$ of $[\![y]\!]$
        such that $[\![y]\!] = f([\![x]\!])$
$\mathtt{OT}\ (0^{k \cdot n}, 0^{k \cdot n}, x_{*3})$
$[y'_{13}, \dots, y'_{m3}] \leftarrow 0^m$
$[y_{13}, \dots, y_{m3}] \leftarrow$
$\mathtt{Reshare}([y'_{13}, \dots, y'_{m3}])$
return $[y_{13}, \dots, y_{m3}]$
```

Figure 4: Detailed algorithms of the hybrid protocol for all computing parties.

instead the tokens for a XOR-gate's output wire are chosen such that the evaluator $\mathcal{CP}_2$ need only perform a bitwise XOR operation on the two input tokens to receive the corresponding output token [KS08].

For non-XOR gates, the garbled row reduction technique applies, meaning that the first row of each non-XOR gate's truth table is not encrypted, but rather, the gate's output tokens are chosen such that $\mathcal{CP}_2$ can obtain the output token corresponding to input tokens $X_a^u|0$ and $X_b^v|0$ directly from those input tokens [PSSW09]. For the remaining three rows of the truth table, $\mathcal{CP}_1$ encrypts the corresponding output tokens using the input wires' tokens and saves them in P. $\mathtt{lsb}(X)$ denotes taking the least significant bit from X and is used to extract the value bit from a wire's token.

After all gates are garbled, $\mathcal{CP}_1$ sends P to $\mathcal{CP}_2$, who will start evaluating the circuit gate-by-gate. The encryption scheme used to encrypt the truth tables is a function $\mathsf{Enc} : \{0,1\}^k \times \{0,1\}^k \times \{0,1\}^\tau \times \{0,1\}^k$ which takes secret tokens A and B and a tweak T to encrypt X, resulting in a ciphertext $\mathsf{Enc}(A, B, T, X)$. Enc is defined as

$$\mathsf{Enc}(A, B, T, X) = \pi(K \parallel T)_{[1:k]} \oplus K \oplus X$$

with

$$K = 2A \oplus 4B$$

where $X, A, B \in \{0,1\}^k$ and $T \in \{0,1\}^\tau$. The function $\pi : \{0,1\}^{k+\tau} \rightarrow \{0,1\}^{k+\tau}$ is a random permutation. $\pi(K \parallel T)_{[1:k]}$ denotes taking the first k bits of the result. In our implementation we use as pseudorandom permutation a fixed-key AES-128 with $k = 80$ and $\tau = 48$. The encryption key for AES is randomly generated and renewed after each circuit evaluation. For the tweak T, we use the gate's index encoded as a 48-bit integer. $2A$ denotes a doubling function which can be implemented in many different ways providing different security guarantees [BHKR13]. We chose *multiplication over finite field $GF(2^k)$* due to it providing the best security guarantees.

Decryption is symmetric and uses the same function Enc. For a non-XOR gate g with input wires a and b, the evaluator takes the value bits of the input tokens $X_a^i|p_a$ and $X_b^j|p_b$ and decrypts the truth table row $P[g, p_a, p_b]$.

After $\mathcal{CP}_2$ has succesfully evaluated the circuit, the result $\overline{[\![y]\!]}$ is shared between $\mathcal{CP}_1$ and $\mathcal{CP}_2$ as $\mathcal{CP}_2$ holds the value bits of the output tokens and $\mathcal{CP}_1$ holds the permutation bits of the output tokens. The XOR of the two provides the actual result since the value bits represent the semantics of the tokens, but are masked with the permutation bits.

The protocol ends with a resharing step to share the output securely between all three computing parties. $\mathsf{Reshare}$ denotes the resharing protocol described in [Bog13].

1.4 Security proof

We have now described the hybrid protocol in detail and will give a proof that the presented protocol is secure. We will use the security proof framework of [Bog13]. Our goal is to prove that the hybrid protocol is *perfectly secure.* To prove perfect security, we will need to show that our protocol is *perfectly simulatable* and ends with a *perfectly secure output resharing protocol* (see Theorem 5 from [Bog13]).

The security framework of [Bog13] is based on the ideal vs real world paradigm. The aim is to prove the security of a multiparty computation protocol by showing that attacks against it in the real world can be transformed into attacks in the ideal world, which are roughly equivalent in terms of resources used and probability of success. This is done by constructing a *simulator* $\mathcal{S}$ which can simulate every real world run of the protocol in the ideal world. Since we are operating in the semi-honest model, we must prove security against a passive adversary who does not actively tamper with the defined protocols, but can corrupt a single computing party to see its inputs and outputs.

We will show for each computing party $\mathcal{CP}_i$ that there exists an efficient non-rewinding simulator $\mathcal{S}_i$ which can simulate all the incoming messages to $\mathcal{CP}_i$. We can prove the existence of such a simulator by showing that all incoming messages of a computing party are independent from the inputs of the other parties and that the messages are distributed according to some known distribution.

We will also use Theorem 4 from [Bog13], which states that a protocol consisting of several perfectly simulatable sub-protocols is also perfectly simulatable if

- The output of each sub-protocol is either the input of another sub-protocol or the output of the main protocol.
- The data dependency graph of sub-protocols is a directed acyclic graph.

If we can prove the perfect simulatibility of our protocol, then to achieve perfect security of the whole protocol, we must simply run a perfectly secure output resharing protocol on the output shares. Intuitively, resharing the output guarantees that the output shares are completely independent of the input shares. We use the standard resharing protocol which is described and proved to be perfectly secure in [Bog13] (Algorithm 1).

We will not discuss the security of the garbling scheme directly, as this is well proven by Bellare et al. [BHKR13] who give exact security bounds for the adverserial advantage.

1.5 Simulatability of Oblivious Transfer

Let us first prove that the oblivious tranfer protocol described in Section 1.3.1 is perfectly simulatable. We will divide the oblivious transfer into two parts:

1. The oblivious choice $\overline{[\![Y]\!]} = \overline{[\![X^0]\!]} \cdot (\overline{[\![1]\!]} - \overline{[\![x]\!]}) + \overline{[\![X^1]\!]} \cdot \overline{[\![x]\!]}$
2. Declassifying $\overline{[\![Y]\!]}$ to $\mathcal{CP}_2$

Note that the oblivious choice part is implemented by using successive calls to Sharemind's secure addition and multiplication protocols. Since addition and multiplication are perfectly simulatable [BNTW12], then the whole oblivious choice is a perfectly simulatable protocol according to Theorem 4 from [Bog13]. Thus we know that there exists a simulator $\mathcal{S}_1$ which can simulate the incoming messages that are sent during the oblivious choice to any computing party. Since there is no incoming communication to $\mathcal{CP}_1$ and $\mathcal{CP}_3$ after the oblivious choice, then $\mathcal{S}_1$ is also the simulator for the whole oblivious transfer protocol for parties $\mathcal{CP}_1$ and $\mathcal{CP}_3$.

For $\mathcal{CP}_2$ however, we must construct a simulator which can additionally simulate the shares of $\overline{[\![Y]\!]}$ sent by $\mathcal{CP}_1$ and $\mathcal{CP}_3$. Let $\mathcal{F}_1$ be the trusted third party who executes the ideal functionality of the oblivious choice and $\mathcal{F}_2$ the trusted third party for the declassification of $\overline{[\![Y]\!]}$ to $\mathcal{CP}_2$. We can construct a simulator $\mathcal{S}_{OT}$ for the whole oblivious transfer protocol by extending $\mathcal{S}_1$. We will construct $\mathcal{S}_{OT}$ in the case where the input contains only one bit x with shares (x_1, x_2, x_3) which means $\mathcal{CP}_2$ must receive a single token $Y = X^x$ with length k. This can trivially be extended to the case where the input is a bit vector with arbitrary length. Let $\mathcal{A}$ be the adversary corrupting $\mathcal{CP}_2$. The simulator construction for $\mathcal{S}_{OT}$ is given in Figure 5.

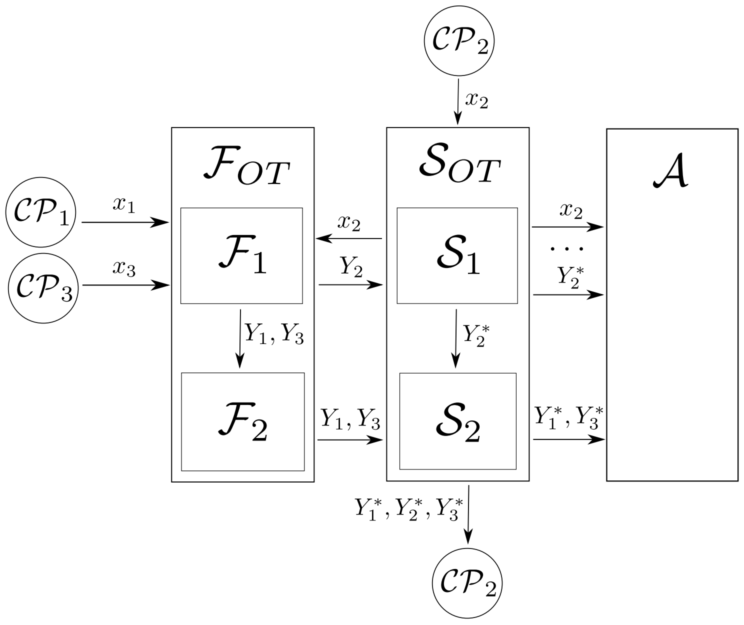

Figure 5: Simulator construction for oblivious transfer.

First, all computing parties $\mathcal{CP}_i$ send their input shares x_i to $\mathcal{F}_1$ who will compute the shares of $[\![Y]\!]$. The adversary also receives x_2, since he sees all incoming messages to $\mathcal{CP}_2$. Let `tokengen`() denote the token generation algorithm that the garbler uses to generate input tokens (Algorithm 3 lines 1-4). Then $\mathcal{F}_1$ generates input tokens $X^0, X^1 \leftarrow$ `tokengen`(), combines the input shares

$x_1 + x_2 + x_3 = x$ and finally calculates $Y = X^x$. $\mathcal{F}_1$ then divides Y into three uniformly distributed shares Y_1, Y_2, Y_3 so that $Y = Y_1 + Y_2 + Y_3$.

The shares Y_1 and Y_3 are simply forwarded to $\mathcal{S}_2$ who will simulate the declassification of $[\![Y]\!]$. $\mathcal{S}_1$ simulates all communication during the oblivious choice to $\mathcal{CP}_2$ and finally sends some $Y_2^* \leftarrow \{0,1\}^k$ to $\mathcal{CP}_2$ which simulates Y_2.

We need $\mathcal{S}_2$ to now simulate the shares Y_1 and Y_3 to $\mathcal{CP}_2$. For this, $\mathcal{S}_2$ can simply replicate the input token generation and oblivious choice performed by $\mathcal{F}_1$. Note that the tokens produced by `tokengen`() are uniformly distributed. $\mathcal{S}_2$ will do the following:

$$\begin{aligned}
X^{0^*}, X^{1^*} &\leftarrow \texttt{tokengen}() \\
x^* &\xleftarrow{\$} \{0,1\} \\
Y^* &\leftarrow (1 - x^*) \cdot X^{0^*} + x^* \cdot X^{1^*} \\
Y_1^* &\xleftarrow{\$} \{0,1\}^k \\
Y_3^* &\leftarrow Y^* - Y_1^* - Y_2^*
\end{aligned}$$

Then $Y_1^* + Y_2^* + Y_3^* = Y^*$, which the adversary cannot distinguish from the actual oblivious choice result Y, since both $Y = X^x$ and $Y^* = X^{x^*}$ are uniformly distributed.

We need to show now that the distribution of all simulated messages in the ideal world coincides with the distribution of messages sent to the adversary in the real world. Since $\mathcal{S}_1$ is a perfect simulator, the messages simulated during the oblivious choice step are identically distributed in the real world. We need to additionally show that this also holds for messages Y_1^* and Y_3^*. We can see from above that Y_1^* and Y_3^* are uniformly and independently distributed. Let us show that this is also the case in the real world for Y_1 and Y_3.

Let us recall how the oblivious choice is calculated for input x and tokens X^0, X^1:

$$\begin{aligned}
[\![\hat{x}]\!] &= [\![1]\!] - [\![x]\!] \\
[\![Y']\!] &= [\![X^0]\!] \cdot [\![\hat{x}]\!] \\
[\![Y'']\!] &= [\![X^1]\!] \cdot [\![x]\!] \\
[\![Y]\!] &= [\![Y']\!] + [\![Y'']\!]
\end{aligned}$$

We can now see, that $Y_1 = Y_1' + Y_1''$ and $Y_3 = Y_3' + Y_3''$ are sums of shares, which are the output shares of products $[\![X^0]\!] \cdot [\![\hat{x}]\!]$ and $[\![X^1]\!] \cdot [\![x]\!]$. These products are calculated using the multiplication protocol of Sharemind, which guarantees that the shares of both $[\![Y']\!]$ and $[\![Y'']\!]$ are uniformly and independently distributed [Bog13]. This means that the shares of $[\![Y]\!]$ are also uniformly distributed and pair-wise independent. Therefore, Y_1 and Y_3 are uniformly and independently distributed and indistinguishable from Y_1^* and Y_3^* to the adversary.

Since the messages simulated by $\mathcal{S}_{OT}$ are distributed identically to the messages sent to the adversary in the real world, then $\mathcal{S}_{OT}$ is a perfect simulator.

1.5.1 Simulatability of the Hybrid Protocol

We will now show that the hybrid protocol as described in Figure 4 is simulatable, ignoring the resharing of the output shares at the end of the protocol. Since we know that the resharing protocol is perfectly secure, it is sufficient to show the simulatability of the protocol without the resharing step in order to prove the perfect security of the whole protocol (Theorem 5 of [Bog13]).

We can see from the detailed algorithm of the whole hybrid protocol on Figure 4 that the only communication that occurs between the computing parties after the oblivious transfer, is sending the circuit's garbled tables P to $\mathcal{CP}_2$. Since there is no incoming communication to $\mathcal{CP}_1$ and $\mathcal{CP}_3$, we need only concern ourselves with the incoming view of $\mathcal{CP}_2$ and must construct a simulator which can simulate the garbled truth tables P.

Note that P is uniquely determined by the circuit C which is being evaluated and the generated input tokens. Therefore, P is computationally independent of the protocol's input (x_1, x_2, x_3), which means that it is trivial to simulate the garbled tables P by simply letting a simulator run Algorithm 3 without the oblivious transfer. However, for perfect simulatability of the whole protocol, we must ensure that we construct the simulator such that the joint distribution of all simulated messages is identical to that in the real world.

For this, we will extend our previous simulator construction with an additional simulator $\mathcal{S}_3$, which will simulate the garbled tables. Let $\mathcal{F}_3$ be the trusted third party who produces the garbled truth tables according to the generated input tokens. The extended simulator construction is presented in Figure 6.

The idea is to simulate the garbled tables by letting the simulator $\mathcal{S}_3$ generate them using the exact same algorithm that the garbler uses in the real world, but using the input tokens generated by $\mathcal{S}_2$. Since $\mathcal{S}_2$ generates the tokens X_0^*, X_1^* using the same algorithm as $\mathcal{F}_1$, then the distribution of the tokens is identical. From this, it follows that since the garbling process itself is completely deterministic given fixed input tokens, then naturally the garbled truth tables P^* produced by $\mathcal{S}_3$ will be identically distributed as P in the real world. Also, the evaluator will be able to evaluate the circuit using Y^* and P^* to receive a valid result, which corresponds to the random input x^* generated by $\mathcal{S}_2$. Therefore, the constructed simulator $\mathcal{S}_H$ is sufficient to prove the simulatability of the whole hybrid protocol and by using the resharing protocol in the end, we can extend simulatability to perfect security.

Now, the only concern left for security is that the adversary should not be able to deduce anything from the garbled tables about the wire tokens which are not explicitly available to $\mathcal{CP}_2$ or the inputs of the other parties. This means that

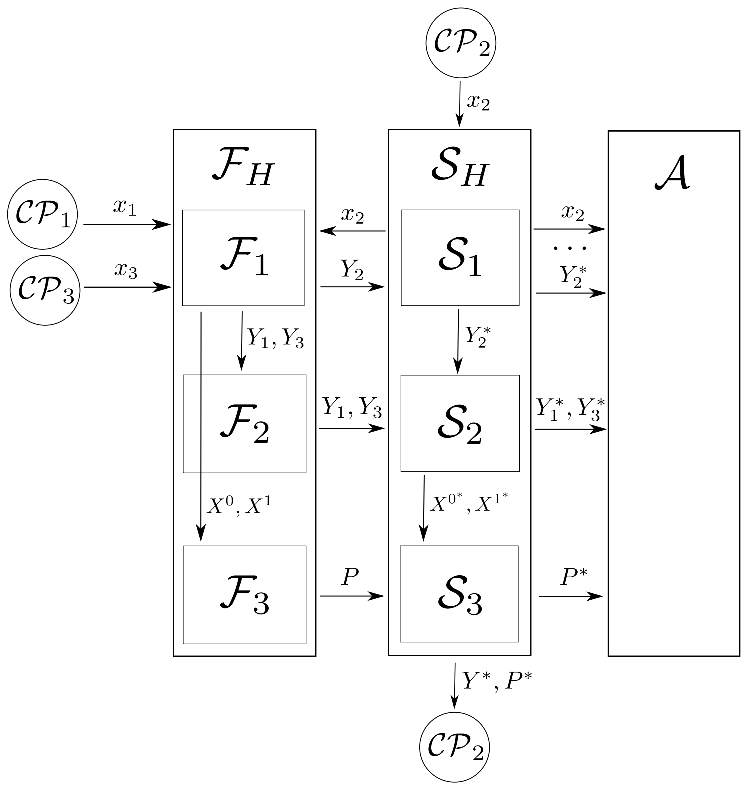

Figure 6: Simulator construction for the whole hybrid protocol.

the evaluator should not be able to decrypt rows of the garbled truth tables for which he does not possess the tokens for, or learn something about the semantics of the wire tokens available to him. These kinds of possible vulnerabilities are direct attacks against the garbling scheme used and are out of the scope of this document. For the garbling scheme used in our protocol, a thorough security analysis for these kinds of vulnerabilites and the bounds for the adverserial advantage are presented in [BHKR13].

2 Implementation details

2.1 Using the protocol to implement primitive operations

Our goal is to use the hybrid protocol in Sharemind to implement a range of composable primitive operations. Current Sharemind protocols are designed to operate on elements of a ring $\mathbb{Z}_{2^n}$ [BNTW12], but such protocols are not well-suited for robust bit level manipulation over data types with arbitrary bit-width. The hybrid protocol is much better suited for implementing such operations, since we have designed it to operate on bitwise secret shared boolean vectors with arbitrary length depending on the circuit used.

Because the hybrid protocol is universally composable, it can be used together with other Sharemind protocols to provide a versatile set of available secure operations to be used in Sharemind applications. The output of one circuit can be used as the input to another circuit or even a different Sharemind protocol. The advantage of this approach is that we can support a wide range of possible secure calculations without having to generate circuits on the fly.

2.2 Circuit setup

Since the hybrid protocol does not generate circuits itself, it requires a circuit description in some suitable format for each primitive that needs to be implemented. Currently we have used circuits from Stefan Tillich and Nigel Smart from the University of Bristol [TS13] and Kreuter et al. [KSS12] for benchmarking our protocol. Also, we have experimented with the new PCF circuit compiler and interpreter [KMSB13] to generate custom circuits from C programs and evaluate them using our hybrid protocol. In all cases, the circuits are stored as individual files on the Sharemind computing party servers. The protocol takes the name of the circuit as an input argument and parses the corresponding circuit file to compute the result. Both the garbler and evaluator parse the circuit exactly the same way since we are not required to hide the function that is evaluated, only the input and output data along with intermediary computation results.

Tillich and Smart's circuits are presented in a straight-forward format which lists all gates in the circuit along with their input and output wires in a topological order. For these circuits, we wrote our own simple circuit parser which reads the whole circuit structure into memory once before evaluation. Since our chosen garbling scheme is not suitable for garbling 1-to-1 gates, our parser optimizes out the logical inverse gates from these circuits. This is achieved by modifying the truth tables of the gates which have inputs originating from an inverse gate. However, since we do not want to lose XOR gates from the circuit, we do not modify their truth tables in case of an inverted input, but change the truth tables of successive non-XOR gates instead. Since evaluating the circuit requires no extra information from the circuit file, the parsing can be done in an offline phase, which reduces the time spent for garbling and evaluating.

However, keeping the whole circuit directly in memory consumes large amounts of memory, which makes this method impractical for very large circuits.

Kreuter et al. were kind enough to provide us with a parser for their circuits, which we used as basis in our protocol. Their circuits are presented in a compact binary format. The parser does not read the whole circuit structure into memory at once, but provides an interface which emits the circuit's gates one-by-one. At any time, only a working set of the circuit's wires are kept in memory. This method scales much better with larger circuits, but introduces a slight performance drawback, since circuits need to be parsed again for each successive evaluation, increasing the time spent for garbling and evaluating.

We also integrated the PCF interpreter with our hybrid protocol. The C implementation of the PCF interpreter can be used as an external library and provides a circuit parsing black box to be used with any secure computation system to handle parsing and evaluating circuits compiled with the PCF compiler. It provides a simple interface which emits circuit gates similarly to Kreuter et al.'s circuit parser. The reading and writing of data to circuit wires is handled via customly definable callbacks, giving the opportunity to use any desired garbling scheme to securely evaluate the circuit.

2.3 Optimizations

We have implemented some standard optimizations for the garbling and evaluating of circuits in the hybrid protocol. As discussed in Section 1.3.2, the garbling scheme we use incorporates the *free-XOR* [KS08] and *garbled row reduction* [PSSW09] optimizations.

The *free-XOR* technique removes the need to garble XOR-gates by choosing the wire tokens in a clever way. Namely, for a XOR-gate, the tokens for the gate's output wire are chosen such that the bitwise XOR of the input tokens always results in the correct output token for the gate according to the semantics of the tokens. This greatly reduces the communication cost for the garbling procedure as no garbled tables need to be sent for XOR-gates. Also, the computational cost for XOR-gates is minimized, since there is no need to encrypt XOR-gates' truth tables.

The *garbled row reduction* method further reduces the communication overhead of garbling by 25%. This is achieved by setting one of the tokens for the output wire of a non-XOR gate as a function of two input wire's tokens. Then for these two input tokens, the corresponding output token can be calculated directly by the evaluator without using the garbled truth table of the gate, effectively reducing the size of the garbled truth table by one row.

We are also using a *streaming* approach to the garbling and evaluating of circuits to parallelize the work of the garbler and the evaluator. When the garbler has finished garbling a batch of the circuit's gates, he can send the garbled truth tables to the evaluator, who can start evaluating the circuit while the garbler encrypts the next batch of gates. The optimal batch size for circuit streaming can be fine-tuned to match the running Sharemind instance's network and hardware capabilities.

3 Experimental results

We now present the performance results of our implemented hybrid protocol prototype. The prototype was benchmarked with various circuits from Stefan Tillich and Nigel Smart [TS13] and Kreuter et al. [KSS12] The descriptions of the used circuits and the corresponding performance results are presented in Tables 1 and 2 respectively.

Table 1 lists for every circuit the size of the input in bits, the overall number of logic gates in the circuit, and the number of XOR-gates in the circuit. The *No of batches* column shows in how many batches the garbled tables are sent from garbler to evaluator. The batch size was fixed for all test runs on 35,000 non-XOR gates' tables.

In Table 2, the performance results for all circuits are presented. The table lists the mean times spent on different phases of the protocol separately, and also the mean total elapsed time. All times are reported in milliseconds with a 95% confidence interval, where 123k denotes 123,000 ms.

The initial parsing time for Kretuer et al.'s circuits is very small due to the fact that the circuit is actually parsed gate-by-gate during garbling and evaluation. Initially, the circuit is memory-mapped so that it could be efficiently parsed during runtime. The runtime parsing time is reflected in the garbling and evaluation times for Kreuter et al.'s circuits.

For Tillich and Smart's circuits, the whole circuit is parsed once before garbling/evaluation. The circuit format is much less efficient to parse than Kreuter et al.'s and therefore the initial parsing times are quite significant. However, in theory the circuit could be parsed once in an offline phase and reused for multiple evaluations, but this approach is not viable if very large circuits need to be evaluated since it would consume large amounts of memory.

The garbling and evaluation times include the time spent on communication in addition to the computational time. Due to network layer instability issues, we were forced to make the garbler thread sleep for 40ms after each batch of garbled tables, except the last, was sent. 40ms was enough so that the next batch would not be sent until the previous had been received by the evaluator. Therefore, the garbling and evaluation times of our prototype are roughly the same for larger circuits. For smaller circuits, the garbling time is almost always smaller, which is due to the communication overhead since evaluation is actually less computationally intensive than garbling.

For the oblivious transfer phase and the total runtime of the protocol, the mean was calculated from the maximum of reported times of all computing parties, meaning that only the reported time of the last computing party to finish was used.

Circuit	Input size(bits)	No of gates	No of XOR-gates	No of batches	Description
Kreuter et al.'s circuits					
mil4	10	57	22	1	Solves the millionaire's problem for 4bit values
mil128	258	1,793	766	1	Solves the millionaire's problem for 128bit values
AES	384	50,935	34,865	1	Encrypts a 128-bit block with given key using AES-128 block cipher
edt-dist128	272	3,442,956	2,007,816	41	Calculates the edit distance of two 128bit strings
dijkstra50	5,216	22,114,948	11,939,325	291	Computes the Dijkstra algorithm on a given 50-node graph
dijkstra100	10,416	168,432,798	90,738,125	2220	Computes the Dijkstra algorithm on a given 100-node graph
Tillich and Smart's circuits					
mult-32x32	64	6,995	1,069	1	Multiplies two 32-bit numbers
AES	256	31,924	25,124	1	Encrypts a 128-bit block with given key using AES-128 block cipher
SHA-256	512	132,854	42,029	3	Calculates the SHA-256 hash of a 512-bit block

Table 1: Descriptions of circuits used for benchmarking the hybrid protocol

Circuit	Initial parsing	Oblivious transfer	Garbling	Evaluation	**Total**
Kreuter et al.'s circuits					
mil4	0.1±0.74%	37.4±1.46%	0.17±0.55%	0.03±0.29%	45.6±0.98%
	0.1±0.73%	37.8±1.44%	0.18±0.53%	0.04±0.41%	45.6±0.97%
mil128	0.2±0.72%	40.0±1.50%	2.3±0.12%	3.2±9.20%	51.0±1.13%
	0.2±0.75%	40.0±1.58%	2.7±0.10%	3.6±9.20%	51.1±1.14%
AES	0.2±1.41%	44.0±1.82%	29.8±0.58%	87.8±0.52%	126.8±0.48%
	0.2±1.47%	44.8±1.70%	34.4±0.59%	92.4±0.51%	132.9±0.55%
edt-dist128	0.2±2.82%	41.0±3.33%	3.88k±0.10%	4.00k±0.11%	4.04k±0.11%
	0.2±2.90%	41.1±3.90%	4.29k±0.11%	4.40k±0.12%	4.44k±0.12%
dijkstra50	2.0±2.59%	124.2±5.38%	27.97k±0.16%	28.05k±0.16%	28.18k±0.18%
	1.9±3.20%	124.8±5.51%	31.34k±0.14%	31.41k±0.14%	31.54k±0.15%
dijkstra100	5.3±1.55%	311.3±4.57%	226.8k±2.75%	226.8k±2.75%	227.2k±2.75%
	5.3±1.82%	326.8±3.62%	252.3k±1.94%	252.3k±1.94%	252.7k±1.94%
Tillich and Smart's circuits					
mult-32x32	37.8±0.34%	38.6±1.53%	12.9±0.40%	30.2±0.86%	104.8±0.46%
	38.3±0.33%	39.2±1.56%	16.3±0.51%	36.0±1.33%	111.3±0.58%
AES	101.4±0.20%	43.1±1.63%	16.1±0.77%	38.3±1.30%	182.5±0.4%
	100.6±0.20%	44.2±1.71%	19.5±0.89%	43.8±0.91%	187.9±0.32%
SHA-256	768.8±0.31%	92.8±5.06%	195.8±0.30%	287.3±0.25%	1,146±0.37%
	778.3±0.27%	94.4±4.51%	224.4±0.27%	305.7±0.29%	1,171±0.32%

Table 2: Performance results of the hybrid protocol in milliseconds with 95% confidence interval. For every circuit, the first row shows results with AES-NI instructions used and the second row with a purely software implemented AES

References

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. *IACR Cryptology ePrint Archive*, 2013:426, 2013.

[BNTW12] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[Bog13] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications.* PhD thesis, University of Tartu, 2013.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Simon [Sim88], pages 1–10.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In Simon [Sim88], pages 11–19.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*. USENIX Association, 2011.

[HKS$^+$10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security. CCS'10*, pages 451–462. ACM, 2010.

[KMSB13] Ben Kreuter, Benjamin Mood, Abhi Shelat, and Kevin Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 321–336, Berkeley, CA, USA, 2013. USENIX Association.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[KSS12] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Towards billion-gate secure computation with malicious adversaries. *IACR Cryptology ePrint Archive*, 2012:179, 2012.

[LP04] Yehuda Lindell and Benny Pinkas. A proof of yao's protocol for secure two-party computation. Cryptology ePrint Archive, Report 2004/175, 2004. `http://eprint.iacr.org/`.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - Secure Two-Party Computation System. In *Proceedings of the 13th USENIX Security Symposium. USENIX'04*, pages 287–302. USENIX, 2004.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. *IACR Cryptology ePrint Archive*, 2009:314, 2009.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[Sim88] Janos Simon, editor. *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*. ACM, 1988.

[TS13] Stefan Tillich and Nigel Smart. Circuits of basic functions suitable for MPC and FHE, 2013.