ABY Developer Guide

Engineering Cryptographic Protocols Group (Encrypto)
TU Darmstadt
www.encrypto.de

May 18, 2018

Contents

1 ABY – Overview 4

1.1 Terminology . . . 6

1.1.1 Sharings . . . 6

1.1.2 Circuits and Gates . . . 7

1.1.3 Wires . . . 7

1.1.4 Shares . . . 7

2 Environment 8

2.1 ABYParty . . . 9

2.1.1 GetSharings . . . 10

2.1.2 GetCircuitBuildRoutine . . . 11

2.1.3 Execution . . . 11

2.1.4 Reset . . . 11

2.1.5 Deletion . . . 11

2.2 Shares . . . 12

2.2.1 Getter Methods . . . 12

2.2.2 Setter Methods . . . 12

2.2.3 Plaintext Access Methods . . . 13

2.2.4 Share Creation . . . 13

2.2.5 Share Bitlengths . . . 14

3 Gates 14

3.1 Input/Output Gates . . . 15

3.1.1 PutINGate . . . 15

3.1.2 PutSharedINGate . . . 16

3.1.3 PutCONSGate . . . 17

3.1.4 PutOUTGate . . . 17

3.1.5 PutSharedOUTGate . . . 18

3.2 Function Gates . . . 18

3.3 Conversion . . . 20

3.4 Debug Gates . . . 21

3.4.1 `PutPrintValueGate` . . . 21

3.4.2 `PutAssertGate` . . . 22

4 SIMD Gates 23

4.1 Changes to Input Gates . . . 24

4.1.1 PutSIMDINGate . . . 24

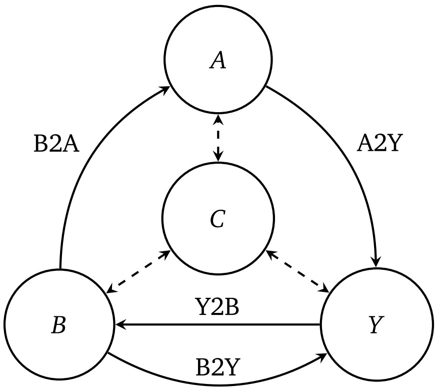
4.1.2 PutSIMDCONSGate 26
4.1.3 PutSIMDAssertGate 27
4.2 Data Management Gates 27
4.2.1 PutCombinerGate 27
4.2.2 PutSplitterGate 28
4.2.3 PutCombineAtPosGate 29
4.2.4 PutSubsetGate 30
4.2.5 PutPermutationGate 31
4.2.6 PutRepeaterGate 32
4.3 Misc Operations 33
4.3.1 get_nvals_on_wire() 33
4.3.2 get_clear_value_vec() 34

5 Benchmarking 34

1 ABY – Overview

ABY [DSZ15] is a framework that allows to use mixed-protocol secure two-party computation protocols, which in turn allow two parties to evaluate functions on sensitive data, while preserving the privacy of this data. The protocols are represented as arithmetic or Boolean *circuits* and can be evaluated privately using arithmetic sharing, Boolean sharing (the GMW protocol) or Yao's garbled circuits. These protocols can also be combined freely, as ABY introduces efficient conversions between them, as depicted in Fig. 1.1.

ABY [DSZ15]是一个框架，它允许使用混合协议安全的两方计算协议，这反过来又允许两方评估敏感数据上的函数，同时保留这些数据的隐私性。这些协议被表示为算术或布尔电路，并且可以使用算术共享、布尔共享(GMW协议)或姚氏乱码电路私下评估。这些协议也可以自由组合，因为ABY在它们之间引入了有效的转换，



ABY框架的概述，它允许在cleartext和三种类型的共享之间进行有效的转换:算术、布尔和Yao。

Figure 1.1: Overview of the ABY framework that allows efficient conversions between **C**leartexts and three types of sharings: **A**rithmetic, **B**oolean, and **Y**ao.

To evaluate a function, ABY represents it internally as an arithmetic or Boolean circuit that consists of potentially many gates. The computing parties then secret share all private values and use arithmetic sharing, Boolean sharing, or Yao sharing to securely evaluate these gates. For a detailed background on the techniques please refer to [DSZ15].

为了计算一个函数，ABY在内部将其表示为一个算术或布尔电路，该电路由可能的许多门组成。计算方然后秘密共享所有私有值，并使用算术共享、布尔共享或Yao共享来安全地评估这些门。

A typical example in secure computation is Yao's millionaires problem, in which two millionaires (here denoted as `SERVER` and `CLIENT`) want to identify who among them is richer without disclosing their actual wealth. In Listing 1.1, we provide a very basic code example which shows every detail that is needed to implement Yao's millionaires problem in ABY using Yao's garbled circuits and also verifying the result of the computation. The corresponding ABY circuit description is depicted in Fig. 1.2 and will serve as a reference example throughout this guide. In the remainder of this guide, we define our terminology (§1.1), detail the main components of ABY (§2), give an outline of the gates that can be used to define the functionality (§3), and outline how to build special SIMD circuits that reduce the memory overhead and improve the computation time (§4).

在清单1.1中，我们提供了一个非常基本的代码示例，其中显示了使用Yao的乱码电路在ABY中实现Yao的百万富翁问题所需的所有细节，并验证了计算结果。相应的ABY电路描述如图1.2所示

Listing 1.1: Simple ABY Code Example for Yao's Millionaires Problem

```
int32_t test_millionaire_prob_simple_circuit(e_role role) {
    //Setup parameters 设置参数
    string address = "127.0.0.1";
    uint16_t port = 6677;
    seclvl seclvl = get_sec_lvl(128);
    e_sharing sharing = S_YAO;
    uint32_t bitlen = 32;
    uint32_t nthreads = 1;
    ABYParty* party = new ABYParty(role, (char*) address.c_str(),
       port, seclvl, bitlen, nthreads);
    vector<Sharing*>& sharings = party->GetSharings();
    Circuit* circ = sharings[sharing]->GetCircuitBuildRoutine();
                                           使用一个名为circuit的电路对象
    //Plaintext values for testing and their bit length 示例数据
    uint32_t alice_money = 5;
    uint32_t bob_money = 7;
    uint32_t money_bitlen = 3;

    //Input shares 数据输入到门限里面                               在circuit上放置了一个高级大于门
    share* s_alice_money = circ->PutINGate(alice_money,            (GTGate)，该大于门在内部由ABY转换成
       money_bitlen, CLIENT);                                      一个由异或门和与门组成的布尔电路。
                                                                   使用PutINGate秘密共享其明文值
    share* s_bob_money = circ->PutINGate(bob_money, money_bitlen,
       SERVER);

    //Greater-than operation 门限里进行比较
    share* s_out = circ->PutGTGate(s_alice_money, s_bob_money);

    //Output share   输出门限的计算值
    s_out = circ->PutOUTGate(s_out, ALL); 使用PutOUTGate重构明文输出

    //Execute secure computation protocol 执行安全计算协议
    party->ExecCircuit();

    //Get plain text output 得到明文输出
    uint32_t output = s_out->get_clear_value<uint32_t>();

    //Verification 进行验证
    cout << "Testing Millionaire's Problem in " <<
       get_sharing_name(sharing) << " sharing: " << endl;
    printf("\nAlice Money:\t %d", alice_money);
    printf("\nBob Money:\t %d", bob_money);
    printf("\nCircuit Result:\t %s" ,(output ? "Alice" : "Bob"));
    printf("\nVerify Result: \t %s\n",((alice_money > bob_money) ?
        "Alice" : "Bob"));

    delete party;
    return 0;
}
```

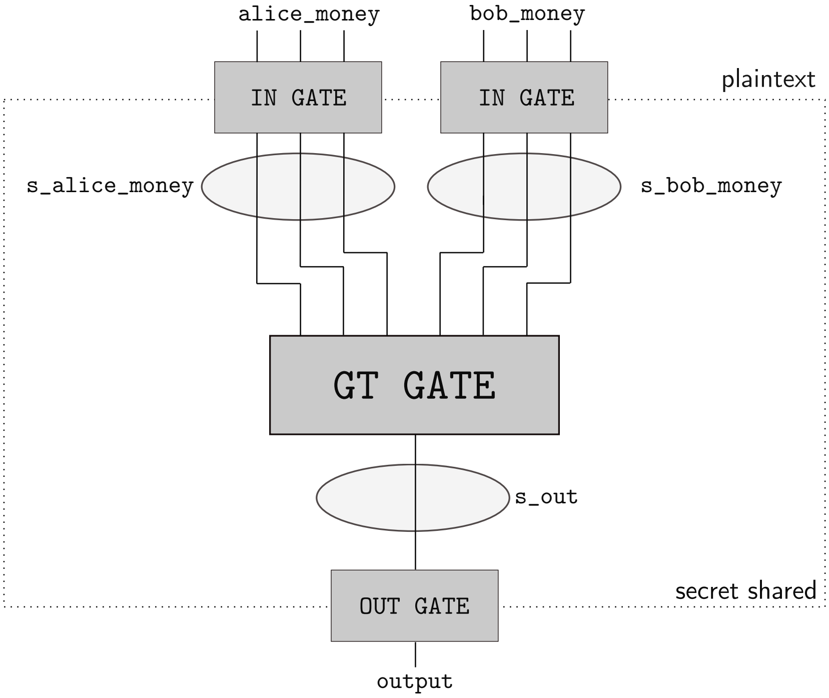


Figure 1.2: Millionaires problem that compares the wealth of Alice and Bob by computing output = alice_money > bob_money the amount of money both parties have has a length of 3 bits and each wire represents a single bit.

1.1 Terminology

In this section, we briefly explain the terminology used in this guide. We refer the reader to [DSZ15] and references therein for details.

1.1.1 Sharings

在ABY中有三种可用的共享:算术共享(S_ARITH)，使用GMW的布尔共享(S_BOOL)，或使用Yao的乱码电路的Yao共享(S_YAO)。

By *sharings* we refer to the underlying secure computation techniques that are used to protect the privacy of the processed data. In ABY there are three sharings available: Arithmetic sharing (S_ARITH), Boolean sharing using GMW (S_BOOL), or Yao sharing using Yao's garbled circuits (S_YAO).

As an example, in Listing 1.1 the plaintext values alice_money and bob_money are secret shared using the S_YAO sharing, from which we obtain secret shared variables s_alice_money and s_bob_money. These variables are hidden (encrypted in some sense) and can only collaboratively be reconstructed to plaintexts.

明文值alice_money和bob_money使用S_YAO共享进行秘密共享，从中我们获得秘密共享变量s_alice_money和s_bob_money。这些变量是隐藏的(在某种意义上是加密的)，只能通过协作将其重构为明文。

1.1.2 Circuits and Gates

ABY允许将函数表示为算术电路或布尔电路。通常，安全计算协议使用两种基本门操作:线性门(算术电路中的加法，布尔电路中的异或)和非线性门(算术电路中的乘法，布尔电路中的与)。线性门可以在不使用简单操作通信的情况下进行局部评估，而非线性门则需要通信和加密操作的评估。因此，安全评估函数的性能可以通过减少非线性门的数量来提高。除了这些基本的门操作之外，ABY还包括用于秘密共享输入、重建输出和内部转换的高级操作的各种门

A secure computation protocol evaluates a function, represented as a *circuit*, on private values by secret sharing them and evaluating the *gates* on the secret shared data. ABY allows to represent a function either as arithmetic or Boolean circuits. Typically, secure computation protocols use two primitive gate operations: *linear gates* (addition in arithmetic circuits, XOR in Boolean circuits) and *non-linear gates* (multiplication in arithmetic circuits, AND in Boolean circuits). While the linear gates can be evaluated locally without communication using simple operations, the non-linear gates require communication and the evaluation of cryptographic operations. Hence, the performance of securely evaluating a function can be improved by reducing the number of non-linear gates. Other than these primitive gate operations, ABY includes various gates for secret sharing inputs, reconstructing outputs, and high-level operations that are translated internally.

In ABY, we use a circuit object called `Circuit` from which we derive two sub-classes called `ArithmeticCircuit` and `BooleanCircuit`. In our example in Listing 1.1, the parties define a `Circuit` object `circ`, on which they put a high-level greater-than gate (`GTGate`), which is internally translated by ABY into a Boolean circuit consisting of XOR and AND gates. Additionally, the parties use the `PutINGate` to secret share their plaintext values and the `PutOUTGate` to reconstruct the plaintext output. These gates are explained in detail in §3.

1.1.3 Wires

The gates in the circuit are connected by *wires* that hold elements in the group $\mathbb{F}_{2^\ell}$. In Boolean or Yao sharing these elements are single bits ($\ell = 1$), while in arithmetic Sharing these elements are char, short, integer or double types ($\ell \in \{8, 16, 32, 64\}$), i.e. all operations are mod 2^ℓ. In ABY, wires are uniquely identified by a global identifier (called wire ID) of type `uint32_t`. This wire ID representation is used mostly internally, while the `share` object abstracts from it and is intended for use in most common operations by the developer.

Remark: With SIMD operations it is also possible to have multiple elements on a single wire simultaneously. This is explained in more detail in §4 and can be ignored for now.

1.1.4 Shares

直观地说，共享可以看作是ABY中的一个变量，它可以传递给门来执行操作，并且可以将门操作的输出分配给它。在执行安全计算协议之后，可以从输出门返回的相应共享对象获得明文值。

To simplify the design of circuits, we abstract from single wires and bundle one or multiple wires in a `share` object. Intuitively, a `share` can be seen as a variable in ABY, which can be passed to gates to perform operations, and which can be assigned the output of a gate operation. After the secure computation protocol has been executed, the plaintext values can be obtained from the respective `share` object, which was returned by an output gate. In our example in Listing 1.1, Alice and Bob obtain secret shares of their plaintext input values in Lines 18 and 19, which they input in the `GTGate` in Line 22 to compute the greater-than operation. Its result is then reconstructed to plaintext using the OUTGate in Line 25, from which we obtain the plaintext value in Line 31.

A share object internally stores an array of uint32_t wire IDs and we refer to the size of this array as the *bitlength* of the share. The wires in a share can be accessed through getter and setter methods in order to operate on these wires directly. However, we recommend accessing single wires only in special cases, since this can become very complex, depending on the functionality.

2 Environment

为了使用ABY框架，必须生成类ABYParty的实例。

In order to use the ABY framework, an instance of the class ABYParty has to be generated. This ABYParty object represents a secure computation party and can be used to obtain methods for defining the functionality and starting the secure computation. Defining the *functionality* that should be evaluated securely is done via interaction with a Circuit object. A Circuit object of a certain sharing type sharing can be obtained from the ABYParty object. Once the functionality has been defined, the secure computation protocol can be performed via the ExecCircuit() function of ABYParty. In this section, we first describe the ABYParty routines (§2.1) and then detail methods for accessing a share (§2.2). We outline gates separately in §3.

某种共享类型的Circuit对象可以从ABYParty对象中获得。一旦定义了功能，就可以通过ABYParty的ExecCircuit()函数执行安全计算协议。

Listing 2.1: ABY Environment Setup

```
int32_t test_millionaire_prob_simple_circuit(e_role role) {
    //Setup
    string address = "127.0.0.1";
    uint16_t port = 6677
    seclvl seclvl = get_sec_lvl(128);
    e_sharing sharing = S_YAO;
    uint32_t bitlen = 32;
    uint32_t nthreads = 1;
                                          生成类ABYParty的实例
    ABYParty* party = new ABYParty(role, (char*) address.c_str(),
       port, seclvl, bitlen, nthreads);
    vector<Sharing*>& sharings = party->GetSharings(); 共享的向量
    Circuit* circ = sharings[sharing]->GetCircuitBuildRoutine();
    // the circuit is defined here...     定义计算的电路

    party->ExecCircuit(); 执行安全计算协议
    uint32_t output = s_out->get_clear_value<uint32_t>();
    // the output can be processed here...     重构明文输出

    delete party;
    return 0;
}
```

The code segment in Listing 2.1 will be used as a reference in the following section, that describes how to set up the *environment*.

2.1 ABYParty

To set up the framework environment, an object of the class ABYParty needs to be instantiated. The instantiated object of ABYParty corresponds to one of the two parties (denoted as SERVER or CLIENT) which participates in the secure computation protocol and receives information about the setup as input parameters.

ABYParty的实例化对象对应于参与安全计算协议的双方(表示为SERVER或CLIENT)中的一方，并接收有关设置的信息作为输入参数。

```
ABYParty(e_role pid, char* addr = (char*) "127.0.0.1", uint16_t
    port = 7766, seclvl seclvl = LT, uint32_t bitlen = 32, uint32_t
     nthreads =  2, e_mt_gen_alg mg_algo = MT_OT, uint32_t maxgates
     = 4000000);
```

Parameters

- role specifies the role of the party, which can be either SERVER or CLIENT. The difference between the SERVER and CLIENT roles is that the SERVER listens for an open connection and acts as circuit garbler in Yao's garbled circuits while the CLIENT connects and acts as circuit evaluator in Yao's garbled circuits.

SERVER和CLIENT角色之间的区别在于，SERVER侦听打开的连接并在Yao的乱码电路中充当电路加成器，而CLIENT在Yao的乱码电路中连接并充当电路评估器。

- address specifies the IP address of the server. If the party acts as server, it opens a socket on this IP address, whereas if the party acts as client, it tries to connect to this IP address.
- port specifies the port to listen on / connect to. The default port is 7766.
- seclvl specifies the security level that is used internally and defaults to long-term security. The required seclvl struct can be returned by providing a symmetric security parameter to get_sec_lvl or by using the respective constant. Available choices are short (ST, 80-bit symmetric security), mid (MT, 112-bit), long (LT, 128-bit), extra-long (XLT, 192-bit), or extra-extra-long (XXLT, 256-bit) term.

Seclvl指定内部使用的安全级别，默认为长期安全。可以通过向get_sec_lvl提供对称安全参数或使用相应的常量来返回所需的seclvl结构。可以选择短(ST，80位对称安全性)、中(MT，112位)、长(LT，128位)、超长(XLT，192位)或超长(XXLT，256位)期限。

- bitlen specifies the bit-length of variables in the arithmetic sharing and can be $\{8, 16, 32, 64\}$. The default value is 32 bits. In Boolean sharing and Yao's garbled circuits, this value is set as the default value for the maximum bitlength of shares.
- nthreads specifies the number of threads that are used in the setup phase.

Nthreads指定在设置阶段使用的线程数。

mg_algo是一个可选参数，可用于更改生成算术乘法三元组的算法(参见[DSZ15]了解详细信息)。可能的选择是:MT_OT、MT_PAILLIER、MT_DGK，即基于OT扩展、Paillier公钥加密方案或damg rd- geisler – kø igaard公钥加密方案生成MT。默认值是MT_OT。

- mg_algo is an optional parameter which can be used to change the algorithm with which arithmetic multiplication triples are generated (see [DSZ15] for details). Possible choices are: MT_OT, MT_PAILLIER, MT_DGK, i.e., MT generation based on OT extension, on the Paillier public-key encryption scheme or on the Damgård-Geisler-Krøigaard public-key encryption scheme. The default is MT_OT.
- maxgates specifies the maximum number of gates which can be built. By default, the value is set to 4 000 000.

Maxgates指定可以构建的门的最大数量。B

Example In Listing 2.1, we instantiate a party (of role server/client) which listens on/-connects to localhost (127.0.0.1) on port 7766 with a long-term security parameter of 128-bits symmetric security, uses internal variables of at most 32-bit length, and uses one thread in the setup phase.

```
ABYParty* party = new ABYParty(role, (char*) address.c_str(), port
    , seclvl, bitlen, nthreads);
```

After the ABYParty object is created, it can be used to execute the defined circuit implementation with code described next.

创建ABYParty对象后，可以使用它来执行已定义的电路实现

2.1.1 GetSharings

In the ABY framework, a sharing describes a secure computation protocol which can be used to define and evaluate a given circuit. The currently supported sharings are: *Arithmetic sharing* (S_ARITH), *Boolean sharing* (S_BOOL) and *Yao sharing* (S_YAO). In order to access such a sharing and define a functionality that should be evaluated, the GetSharings() method of the ABYParty class is invoked.

在ABY框架中，共享描述了一种安全的计算协议，该协议可用于定义和评估给定电路。目前支持的共享有:算术共享(S_ARITH)、布尔共享(S_BOOL)和Yao共享(S_YAO)。为了访问这样的共享并定义应该评估的功能，调用ABYParty类的GetSharings()方法。

```
vector<Sharing*>& ABYParty::GetSharings();
```

This method returns a vector of all the supported sharings by the framework.

Example In Listing 2.1, in line 10, we obtain the vector of sharings and store it in the variable sharings.

```
vector<Sharing*>& sharings = party->GetSharings();
```

2.1.2 GetCircuitBuildRoutine

为了定义要评估的功能，需要获得期望共享的相应电路对象。该Circuit对象可以在指定的共享节点2 fS_ARITH, S_BOOL, S_YAOg上通过getcircuitbuiloutine()获取。

To define the functionality that should be evaluated, one needs to obtain a corresponding Circuit object of the desired sharing.

This Circuit object can be obtained using the GetCircuitBuildRoutine() on the desired sharing ∈ {S_ARITH, S_BOOL, S_YAO}. Note that, if sharing is S_ARITH, the circuit is of type ArithmeticCircuit, while for S_BOOL and S_YAO the circuit is of type BooleanCircuit.

```
Circuit* circ = sharings[sharing]->GetCircuitBuildRoutine();
```

2.1.3 Execution

After the environment has been set up as described above, the actual circuit consisting of several gates has to be constructed. We detail high-level gates that can be used to define the function in more detail in §3. After the circuit has been built, it is evaluated securely by invoking the ExecCircuit() method.

电路构建完成后，将通过调用ExecCircuit()方法安全地对其进行评估。

```
party->ExecCircuit();
```

2.1.4 Reset

After the circuit has been evaluated, the ABYParty object can be re-used to build a second circuit to evaluate securely. In this case, the Reset method has to be invoked before building the second circuit in order to reset the internal state. Network connections remain open and some initializations and allocations are re-used, thus saving time compared to a full restart of the entire framework.

在对电路进行评估之后，可以重用ABYParty对象来构建第二个电路以安全地进行评估。在这种情况下，必须在构建第二个电路之前调用Reset方法，以便重置内部状态。网络连接保持打开状态，一些初始化和分配被重用，因此与完全重新启动整个框架相比节省了时间。

```
party->Reset();
```

2.1.5 Deletion

在程序终止之前，应该通过以下方式释放party对象：

The party object should be deallocated before the program terminates via:

```
delete party;
```

When handling huge implementations with the ABY Framework, it is recommended to deallocate the ABYParty object after its scope has been terminated.

2.2 Shares

共享对象被用作ABY框架中的变量。

The share objects are used as variables within the ABY framework. In the following section, we detail several methods that can be used to access and modify the internal state of shares.

2.2.1 Getter Methods

Each wire has a unique wire ID of type uint32_t. The wires in a share are stored internally as vector<uint32_t>. These wires can be read using the methods: get_wire_id(), get_wire_ids(), or get_wire_ids_as_share().

共享中的连线在内部存储为vector。可以使用get_wire_id()、get_wire_ids()或get_wire_ids_as_share()方法读取这些连线。

get_wire_id() Returns a single wire at position posid.

```
uint32_t get_wire_id(uint32_t posid);
```

get_wire_ids() Returns a list of wires stored in a share as vector<uint32_t>.

```
vector<uint32_t> get_wire_ids();
```

get_wire_ids_as_share() Returns a share object of the wire at position posid.

```
share* get_wire_ids_as_share(uint32_t posid);
```

2.2.2 Setter Methods

存储在共享中的内部连线可以使用set_wire_ids()或set_wire_id()方法写入。通过设置导线，可以将一个门的输出子集指定为另一个门的输入，或者在不评估门的情况下执行位置换。

Internal wires that are stored in the share can be written using the methods: set_wire_ids() or set_wire_id(). By setting wires one can assign a subset of the outputs of a gate as input to another gate or perform bit permutations without evaluating gates.

set_wire_ids() Sets the wires in a given share object to the provided wireids.

```
void set_wire_ids(vector<uint32_t> wireids);
```

set_wire_id() Sets the wire at position posid to the wire with ID wireid.

```
void set_wire_id(uint32_t posid, uint32_t wireid);
```

2.2.3 Plaintext Access Methods

共享存储了保存秘密共享值的连接。在使用execCircuit执行安全计算协议之后，可以使用get_clear_value()方法从共享对象中检索输出线上的明文值(参见§3.1.4中的PutOutGate)。可以使用模板T来改变输出类型

A share stores wires which again hold secret-shared values. After the secure computation protocol has been executed using execCircuit, plaintext values on the output wires (see PutOutGate in §3.1.4) can be retrieved from share objects using the get_clear_value() method. The output type can be varied using a template T.

```
template<class T> T get_clear_value();
```

Example The following example illustrates how to use get_clear_value() to obtain the plaintext output of the share s_out as uint32_t.

```
uint32_t output = s_out->get_clear_value<uint32_t>();
```

还有get_clear_value_ptr()，它返回一个指向明文结果的指针。

There is also get_clear_value_ptr(), that returns a pointer to a plaintext result. It is used when the result does not fit into a standard data type, typically, if it is longer than 64 bits.

```
uint8_t *output = s_out->get_clear_value_ptr();
```

2.2.4 Share Creation

为了创建共享对象，可以使用Circuit对象秘密共享明文输入值(参见§3.1.1中的PutInGate)，或者使用下面描述的静态create_new_share()方法初始化对象并从头手动分配电线。

In order to create a share object, one can either secret-share a plaintext input value (see PutInGate in §3.1.1) using a Circuit object or initialize the object and manually assign wires from scratch using the static create_new_share() methods described in the following.

```
1 static share* create_new_share(uint32_t size, Circuit* circ);
2 static share* create_new_share(vector<uint32_t> wireids, Circuit*
      circ);
```

Initializes and returns a pointer to a share object. The first option (Line 1) creates an empty share object to which wires need to be assigned using the setter methods before it can be used as an input to a gate. The second option (Line 2) creates the share from the vector of wire IDs.

初始化并返回指向共享对象的指针。第一个选项(第1行)创建一个空的共享对象，需要使用setter方法将连接分配给该对象，然后才能将其用作门的输入。第二个选项(第2行)从连接id向量创建共享。

Parameters

- size the number of wires that can be stored in the share, initialized to zero.
- wireids the wires that should be stored in the share.
- circ the Circuit object from which the wires come.

设置共享中可以存储的连接数的大小，初始化为零。
连接应该存储在共享中的电线。
将电线来自的Circuit对象圈起来。

2.2.5 Share Bitlengths

共享保存特定位长的值，并在内部存储允许值增长到的最大位长。可以通过get_bitlength()、set_bitlength()、get_max_bitlength()和set_max_bitlength()方法访问共享的比特长度和最大比特长度。默认情况下，每个共享的最大比特长度被设置为ABYParty对象中传递的比特长度(参见§2.1)。

Shares hold values of a specific bitlength and internally store the maximum bitlength to which a value is allowed to grow. The bitlength of a share and the maximum bitlength that it can support can be accessed via the following methods: `get_bitlength()`, `set_bitlength()`, `get_max_bitlength()`, and `set_max_bitlength()`. By default, the maximum bitlength of each share is set to the bitlength that is passed in the `ABYParty` object (see §2.1).

get_bitlength() Returns the current bitlength of the share.

```
uint32_t get_bitlength();
```

set_bitlength() Sets the bitlength of the current share to `bitlength`. Note, however, that this routine does not add any wires. Using such a share will cause errors if used in a gate unless wires are added by the developer via `set_wire()`. Changing the bitlength only works for Boolean or Yao sharing, but not for arithmetic sharing.

```
void set_bitlength(uint32_t bitlength);
```

get_max_bitlength() Returns the maximum bitlength up to which values in this share can grow.

```
uint32_t get_max_bitlength();
```

set_max_bitlength() Sets the maximum bitlength to which values in this share can grow to `max_bitlength`, which must be at least as big as the current bitlength of the share. Changing the maximum bitlength only works for Boolean sharing or Yao sharing, but not for arithmetic sharing. For Boolean circuit-based protocols, the bitlength of the results from operations such as addition or multiplication will grow up to `max_bitlength` and larger results will be truncated to `max_bitlength`.

```
void set_max_bitlength(uint32_t max_bitlength);
```

3 Gates

The ABY framework implements several secure computation protocols that operate on arithmetic or Boolean circuits. In the previous chapter §1.1.1, we have described how values are secret shared between two parties and internally represented as wires in a circuit. In this chapter, we explain how values on these wires can be processed using Gates. In other words, gates are used for executing various operations on the provided user inputs.

门用于对提供的用户输入执行各种操作。

These Gates can be created in ABY using Circuit objects (§1.1.2). We first describe *I/O gates* (§3.1) that transform plaintext values into secret-shared values and back to recover the plaintext value from a share. Next, we describe *Function gates* (§3.2) that compute on secret shared values using. Then, we describe how to convert secret shared values between different secure computation schemes using *Conversion gates* (§3.3). Finally, we describe *Debug* gates that ease the development process (§3.4).

3.1 Input/Output Gates

输入和输出门用于明文值和隐藏(加密/秘密共享)值之间的转换和返回。输入门(PutINGate， 见§3.1.1)接受明文值并执行加密/秘密共享操作来创建一个可以用函数门处理的共享对象。恒定门(putcongate， 见§3.1.3)可用于向电路输入双方都知道的恒定值。在份额计算完成后，加密/秘密共享操作可以通过输出门(PutOUTGate，参见§3.1.4)撤消，该输出门将产生的份额转换回明文值。

Input and output gates are used for the transition between plaintext values and hidden (encrypted/secret-shared) values and back. Input gates (PutINGate, see §3.1.1) take plaintext values and perform an encryption/secret-sharing operation to create a share object that can be processed with function gates. Constant gates (PutCONSGate, see §3.1.3) can be used to input constant values to the circuit that are known by both parties. After computations on shares are finished, the encryption/secret-sharing operation can be undone with an output gate (PutOUTGate, see §3.1.4), which transforms the resulting shares back to plaintext values. ABY also allows to input pre-shared values (PutSharedINGate, see §3.1.2) and output the secret shares of a value (PutSharedOUTGate, see §3.1.5).

3.1.1 PutINGate

PutInGate用于将各方的明文输入加载到它们的共享中。

PutInGate is used to load the plaintext inputs of the respective parties to their shares.

```
share* PutINGate(uint64_t val, uint32_t bitlen, e_role role);
share* PutINGate(uint32_t val, uint32_t bitlen, e_role role);
share* PutINGate(uint16_t val, uint32_t bitlen, e_role role);
share* PutINGate( uint8_t val, uint32_t bitlen, e_role role);

share* PutINGate(uint64_t* val, uint32_t bitlen, e_role role);
share* PutINGate(uint32_t* val, uint32_t bitlen, e_role role);
share* PutINGate(uint16_t* val, uint32_t bitlen, e_role role);
share* PutINGate( uint8_t* val, uint32_t bitlen, e_role role);
```

The method returns a share object holding a share or encryption of a plaintext value.

该方法返回一个共享对象，该对象持有明文值的共享或加密。

Parameters

- val is the input value loaded by one of parties to generate the shared secret. This variable can be of type uint8_t, uint16_t, uint32_t, uint64_t, or a pointer to an array of such a datatype.
- bitlen states how many bits of the plaintext should be read and specifies the number of wires in the generated share.
- role defines which party provides the input value for this share. The role value can be either CLIENT or SERVER.

Example

```
//Share creation
uint32_t alice_money = 5;
uint32_t bob_money = 7;
uint32_t bitlen = 3;

//Input
share *s_alice_money, *s_bob_money;
s_alice_money = circ->PutINGate(alice_money, bitlen, CLIENT);
s_bob_money = circ->PutINGate(bob_money, bitlen, SERVER);
```

alice_money and bob_money are the plaintext inputs for their respective shares. The inputs are values with a length of 3 bits, which are indicated by the input bitlen. The third parameter provided is the role of the party which provides the input to the share. The value of role can be CLIENT or SERVER, depending on the role played by the party.

Note that, although the inputs for *both* roles have to be defined, every party can set only its own input values and receives the values of the other party encrypted or secret-shared. This also means, that the input values of the respective other party are ignored and can be set to zero or an arbitrary value. More specifically, both parties need to specify the existence of *all* inputs and who provides the corresponding plaintext input. The actual plaintext value needs to be provided only by the respective party. The PutDummyINGate can be used to explicitly mark an input of the other party without providing plaintext input values. It has the same interface as the PutINGate but without the plaintext value val.

请注意，尽管必须定义两个角色的输入，但每一方只能设置自己的输入值，并接收加密或秘密共享的另一方的值。这也意味着，对方的输入值将被忽略，并且可以设置为零或任意值。更具体地说，双方都需要指定所有输入的存在性以及谁提供相应的明文输入。实际的明文值只需要由各自的一方提供

The SIMD version of the same gate is discussed in the SIMD Gates chapter §4.

3.1.2 PutSharedINGate

PutSharedINGate is used to load pre-shared inputs of the parties to shares. Both parties need to provide an input and hence the source role is omitted. These gates are used for values that come from a third party that secret shares them to outsource computation (sometimes referred to as client/server model), or for intermediate values from a previous computation.

PutSharedINGate用于将各方的预共享输入加载到共享中。双方都需要提供输入，因此省略了源角色。这些门用于来自第三方的值，这些值被秘密地共享给外包计算(有时称为客户机/服务器模型)，或者用于来自先前计算的中间值。

```
share* PutSharedINGate(uint64_t val, uint32_t bitlen);
share* PutSharedINGate(uint32_t val, uint32_t bitlen);
share* PutSharedINGate(uint16_t val, uint32_t bitlen);
share* PutSharedINGate( uint8_t val, uint32_t bitlen);

share* PutSharedINGate(uint64_t* val, uint32_t bitlen);
share* PutSharedINGate(uint32_t* val, uint32_t bitlen);
share* PutSharedINGate(uint16_t* val, uint32_t bitlen);
share* PutSharedINGate( uint8_t* val, uint32_t bitlen);
```

The method returns a share object for the pre-shared value val of bit-length bitlen. PutSharedINGate can be used together with PutSharedOUTGate (see §3.1.5) to pass shared values between different ABY executions.

该方法返回一个共享对象，其预共享值为位长度为bitlen的val。PutSharedINGate可以与PutSharedOUTGate(参见§3.1.5)一起使用，在不同的ABY执行之间传递共享值。

Remark: Currently, PutSharedINGate only works for arithmetic and Boolean sharing.

3.1.3 PutCONSGate

putcongate函数可用于向电路中输入一个恒定的明文值。Val具有比特长度bitlen，这是双方都知道的。该函数返回一个share对象，它表示秘密共享/加密常量。

The PutCONSGate function can be used to input a constant plaintext value val into the circuit. val has bit-length bitlen, which is known to both parties. The function returns a share object, which represents the secret-shared/encrypted constant.

```
share* PutCONSGate(uint64_t val, uint32_t bitlen);
```

3.1.4 PutOUTGate

PutOUTGate用于交互式地解密共享或将秘密共享值重构为明文，并将结果存储在共享中。调用ExecCircuit()方法后，可以使用get_clear_value()访问明文值。这种访问只能由角色中定义的一方进行。

PutOUTGate is used to interactively decrypt a share or reconstruct a secret-shared value to plaintext and stores the result in a share. After the ExecCircuit() method has been called, the plaintext values can be accessed using get_clear_value(). This access is only possible by the party defined in role.

```
share* PutOUTGate(share* s_out, e_role role)
```

Parameters

- s_out is the share object which should be output after the circuit is evaluated. The plaintext value is extracted and converted into the specified datatype by the respective party using the get_clear_value() methods.
- role defines the party which is allowed to see the output value. The role value can be CLIENT, SERVER, or ALL. 角色定义了允许看到输出值的一方。角色取值包括CLIENT、SERVER和ALL。

Example

```
s_out = circ->PutOUTGate(s_val, ALL);
```

In the above example, we assume the share object s_val contains a computed result. The output gate performs a recombination operation that generates a plaintext value of the computation result and stores it in s_out. In this example both parties can access the computation result after the circuit execution using get_clear_value() (see §2.2.3).

3.1.5 PutSharedOUTGate

PutSharedOUTGate用于将共享的秘密共享值输出到一个可以在电路外部使用的变量中。双方都接收共享输出，因此目标角色作为参数省略。

PutSharedOUTGate is used to output the secret shared value of a share into a variable that can be used outside of the circuit. Both parties receive a shared output and hence the destination role is omitted as parameter.

```
share* PutSharedOUTGate(share* s_out)
```

The method returns a share object for the pre-shared value. PutSharedOUTGate can be used together with PutSharedINGate (see §3.1.2) to pass shared values between different ABY executions.

该方法返回预共享值的共享对象。PutSharedOUTGate可以与PutSharedINGate(见§3.1.2)一起使用，在不同的ABY执行之间传递共享值。

Remark: Currently, PutSharedOUTGate only works for arithmetic and Boolean sharing.

3.2 Function Gates

Function gates are used to compute on shares. The ABY framework includes several standard operations, listed in Tab. 3.1. Note that the operations AND (∧), OR (∨), XOR (⊕), MUX and GT (>) are only available for Boolean circuits (i.e., in Boolean and Yao sharing) and not for arithmetic circuits. We provide an overview of the operations in the following.

请注意，AND（ˆ）， OR（_），XOR(⊕)，MUX和GT(>)操作仅适用于布尔电路(即布尔和Yao共享)，而不适用于算术电路。

Table 3.1: Operations

Operations	AND	XOR	OR	ADD	MUL	SUB	GT	MUX	INV
Arithmetic	✗	✗	✗	✓	✓	✓	✗	✗	✓
Boolean	✓	✓	✓	✓	✓	✓	✓	✓	✓
Yao	✓	✓	✓	✓	✓	✓	✓	✓	✓

PutANDGate PutANDGate performs a bitwise AND operation on the two input shares and returns the result as a share object of the same bitlength as the longer input share.

```
share* PutANDGate(share* ina, share* inb);
```

PutANDGate对两个输入共享执行位与操作，并将结果作为与较长的输入共享具有相同位长的共享对象返回。

PutXORGate `PutXORGate` performs a bitwise XOR operation on the two input shares and returns theresult as a share object of the same bitlength as the longer input share.

```
share* PutXORGate(share* ina, share* inb);
```

PutORGate `PutORGate` performs a bitwise OR operation on the two input shares and returns the result as a share object of the same bitlength as the longer input share.

```
share* PutORGate(share* ina, share* inb);
```

PutADDGate `PutADDGate` performs an arithmetic addition operation on the two input shares and returns the result as a share object. In arithmetic circuits the addition is carried out modulo 2^{ℓ}, where ℓ is the bitlength of the sharing. In Boolean circuits, the result of the addition also includes the carry bit in case the bitlength of the result does not exceed the maximum bitlength of both shares (see §2.2.5 on how to get and set the maximum bitlength of shares).

在算术电路中，加法是对2 ’取模进行的，其中’是共享的位长。在布尔电路中，如果结果的位长不超过两个份额的最大位长，则加法的结果也包括进位

```
share* PutADDGate(share* ina, share* inb);
```

PutMULGate `PutMULGate` performs an arithmetic multiplication operation on the two input shares and returns the result as a share object. In arithmetic circuits the multiplication is carried out modulo 2^{ℓ}, where ℓ is the bitlength of the sharing. In Boolean circuits, the bitlength of the output is the smaller value of either: 1) the sum of both inputs' bitlengths or 2) the largest maximum bit length of the inputs.

```
share* PutMULGate(share* ina, share* inb);
```

PutSUBGate `PutSUBGate` performs an arithmetic subtraction operation on the two input shares and returns the result `ina−inb` as share object. In arithmetic circuits the subtraction is carried out modulo 2^{ℓ}, where ℓ is the bitlength of the sharing. Thus the result is always positive. In Boolean circuits, the subtraction is carried out modulo 2^{max}, where max is the larger maximum bitlength of either `ina` or `inb`.

```
share* PutSUBGate(share* ina, share* inb);
```

PutGTGate对两个输入共享ina和inb执行大于操作(>)，如果ina > inb返回1，否则返回0。

PutGTGate `PutGTGate` performs an greater-than operation (>) on the two input shares `ina` and `inb` and returns 1 if `ina` > `inb` and 0 otherwise.

```
share* PutGTGate(share* ina, share* inb);
```

PutMUXGate实现了一个多路复用器，并根据选择位返回两个给定数据输入中的一个。如果选择位sel为1，则返回ina的内容。如果sel为0，则返回inb。

PutMUXGate PutMUXGate implements a multiplexer and returns one of two given data inputs based on a selection bit. If the selection bit sel is 1, the content of ina is returned. If sel is 0, inb is returned.

```
share* PutMUXGate(share* ina, share* inb, share* sel);
```

PutINVGate The PutINVGate function inverts an input value in. In arithmetic circuits, the inversion is performed by computing $2^{\ell}-$in, while in Boolean circuits the inversion is performed bit-wise by computing $\forall_{i\leq\ell} 1\oplus$in[$i$], where ℓ is the bitlength of in and in[i] refers to the i-th bit of in.

```
share* PutINVGate(share* in);
```

3.3 Conversion

ABY框架允许使用算术、布尔或Yao共享执行安全计算，并使用转换门在它们之间任意转换秘密共享值。

The ABY framework allows to perform secure computation using arithmetic, Boolean or Yao sharing and to arbitrarily convert secret-shared values between them using *Conversion* gates. Unlike the function gates, introduced in §3.2, conversion gates do not change the secret-shared value. Instead, conversion gates transform the shares, held by each of the parties, from the representation of one secure computation scheme into another secure computation scheme. Like all previous operations, the conversion is also done in a way that reveals nothing about the plaintext.

与前面的所有操作一样，转换的方式也不会透露任何关于明文的信息。

In the following, we use the short notation A2B to denote that a method converts a share from arithmetic to Boolean sharing. Given the existing schemes, this gives us six possible conversion methods: A2B, A2Y, B2A, B2Y, Y2A, and Y2B. Note that only four of these methods are implemented: A2Y, B2A, B2Y, and Y2B, as depicted in Fig. 1.1. The remaining two methods, namely A2B and Y2A are implemented by computing Y2B(A2Y) and B2A(Y2B), respectively.

我们使用简短表示法A2B表示一个方法将共享从算术共享转换为布尔共享。鉴于现有的方案，这给了我们六种可能的转换方法：A2B, A2Y, B2A, B2Y, Y2A和Y2B。注意，这些方法中只有四个实现了：A2Y、B2A、B2Y和Y2B

Note that the conversion gate function needs to be invoked from a Circuit of the *target sharing*, i.e., the A2Y function would need to be called on a Circuit for Yao sharing.

A2Y The A2Y function converts an arithmetic share in into a Yao share. The returned share is a Yao share, that has the same plaintext value as the input arithmetic share. Note that the A2Y function needs to be called on a Circuit in Yao sharing.

A2Y函数将算术共享转换为Yao共享。返回的共享是Yao共享，与输入的算术共享具有相同的明文值。

```
share* A2Y(share* in);
```

Example The following example secret shares two 32-bit numbers *A* and *B* in the arithmetic sharing and multiplies them. The product is converted to Yao sharing and again multiplied by two.

```
share *s_a, *s_b, *s_res;
Circuit* ac = sharings[S_ARITH]->GetCircuitBuildRoutine();
Circuit* yc = sharings[S_YAO]->GetCircuitBuildRoutine();
s_a = ac->PutINGate(A, 32, SERVER);
s_b = ac->PutINGate(B, 32, CLIENT);
s_res = ac->PutMULGate(s_a, s_b);
s_res = yc->PutA2YGate(s_res);
s_res = yc->PutADDGate(s_res, s_res);
```

B2A / B2Y / Y2B Similarly to A2Y, after creating the respective circuits and shares, the other conversions can be performed using the following functions: B2A, B2Y, and Y2B.

A2B和Y2A转换是通过顺序调用两个原语转换来执行的:A2B(x) = Y2B(A2Y(x))和Y2A(x) = B2A(A2B(x))。与原始转换相比，A2B和Y2A门需要中间转换电路作为附加参数。也就是说，A2B转换需要S_YAO的电路，而Y2A转换需要S_BOOL的电路。

A2B / Y2A The A2B and Y2A conversions are performed by invoking two primitive conversions in sequence: A2B(x) = Y2B(A2Y(x)) and Y2A(x) = B2A(A2B(x)). In contrast to the primitive conversions, the A2B and Y2A gates require the circuit of the intermediate conversion as additional parameter. I.e., the A2B conversion requires a circuit from S_YAO while the Y2A conversion requires a circuit from S_BOOL.

```
share* A2B(share* in, Circuit* yaosharingcircuit);
share* Y2A(share* in, Circuit* boolsharingcircuit);
```

3.4 Debug Gates

Debugging applications that use secure computation is a tedious task since the intermediate values are secret-shared and hence not easily verifiable. In the following, we outline two gates that support the development process of secure computation applications: the PrintValueGate and the AssertGate. Note that both gates should only be used during the development process and not when private data is processed, since they leak intermediate values. In order to deactivate the Debug gates, set the macro ABY_PRODUCTION to 1 in src/abycore/ABY_utils/ABYconstants.h:25.

3.4.1 PutPrintValueGate

PutPrintValueGate can be used to print the plaintext value of the share object in during the secure function evaluation together with a pre-defined string infostring. Note that the output will be printed once the gate is evaluated, hence the order in which the gates are printed during function evaluation can vary from the order in which they were built during circuit construction.

```
share* PutPrintValueGate(share* in, string infostring);
```

Example The following example prints the sum and the logical and of the two 3-bit numbers alice_money and bob_money. Note that if a Boolean circuit is used, the logical AND will be printed before the sum since ABY schedules the gates depending on their multiplicative (AND) depth.

```
//Share creation
uint32_t alice_money = 5;
uint32_t bob_money = 7;
uint32_t bitlen = 3;

//Input
share *s_alice_money, *s_bob_money, *s_and, *s_add;
s_alice_money = circ->PutINGate(alice_money, bitlen, CLIENT);
s_bob_money = circ->PutINGate(bob_money, bitlen, SERVER);

s_add = circ->PutADDGate(s_alice_money, s_bob_money);
circ->PutPrintValueGate(s_add, "Sum");

s_and = circ->PutANDGate(s_alice_money, s_bob_money);
circ->PutPrintValueGate(s_and, "Logical AND");

party->ExecCircuit();
```

3.4.2 PutAssertGate

PutAssertGate验证share in的明文值等于一个bit -bit明文值assert_val。如果验证成功，程序继续;如果验证失败，程序停止。明文输入assert_val的格式与PutINGate的格式相同

PutAssertGate verifies that the plaintext value of a share in is equal to a bitlen-bit plaintext value assert_val. If the verification succeeds, the program continues and if the verification fails, the program stops. The format of the plaintext input assert_val is the same as for the PutINGate (see §3.1.1).

```
share* PutAssertGate(share* in, uint64_t assert_val, uint32_t
    bitlen);
share* PutAssertGate(share* in, uint32_t assert_val, uint32_t
    bitlen);
share* PutAssertGate(share* in, uint16_t assert_val, uint32_t
    bitlen);
share* PutAssertGate(share* in,  uint8_t assert_val, uint32_t
    bitlen);

share* PutAssertGate(share* in, uint64_t* assert_val, uint32_t
    bitlen);
```

```
share* PutAssertGate(share* in, uint32_t* assert_val, uint32_t
    bitlen);
share* PutAssertGate(share* in, uint16_t* assert_val, uint32_t
    bitlen);
share* PutAssertGate(share* in,  uint8_t* assert_val, uint32_t
    bitlen);
```

Example The following example adds the values alice_money and bob_money and verifies that the secret-shared output s_sum equals the result sum_money that is computed in plaintext.

下面的示例将值alice_money和bob_money相加，并验证秘密共享的输出s_sum是否等于明文计算的结果sum_money。

```
//Share creation
uint32_t alice_money = 5;
uint32_t bob_money = 7;
uint32_t bitlen = 3;
uint32_t sum_money = alice_money + bob_money;

//Input
share *s_alice_money, *s_bob_money, *s_add;
s_alice_money = circ->PutINGate(alice_money, bitlen, CLIENT);
s_bob_money = circ->PutINGate(bob_money, bitlen, SERVER);

s_add = circ->PutADDGate(s_alice_money, s_bob_money);
circ->PutAssertGate(s_add, sum_money, bitlen+1);
```

验证秘密共享的输出s_sum是否等于明文计算的结果sum_money

4 SIMD Gates

SIMD(单指令多数据)表示使用单个操作处理多个数据元素的操作。它是并行编程的一个概念，可用于安全计算，以减少电路的内存占用并提高电路评估时间。

SIMD (Single Instruction Multiple Data) denotes operations that process multiple data elements using a single operation. It is a concept from parallel programming that can be adapted to secure computation to reduce the memory footprint of the circuit and improve the circuit evaluation time.

In the previous chapters, all operations on a share were non-SIMD. In particular, a share was a one-dimensional array that internally stored multiple wires, each of which held a single element in $\mathbb{F}_{2^\ell}$. For the SIMD operations in this chapter, we extend the shares to a *second dimension* by allowing multiple elements in $\mathbb{F}_{2^\ell}$ to be stored on a wire. We give an example in Fig. 4.1, where we depict a two-dimensional Boolean circuit share s_val with $\sigma = 4$ wires (bits) and where each wire stores nvals $= 3$ elements. Note that in the figures in this chapter, the wires are ordered such that the least significant bit is on the rightmost wire while the most significant bit is on the leftmost wire. SIMD values are ordered from top to bottom, i.e., the value at index 0 is displayed at the top and increasing indices follow below.

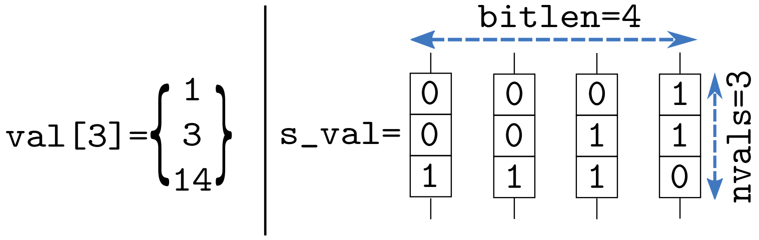


Figure 4.1: SIMD representation example for a Boolean circuit `share` with 4 wires and where each wire stores 3 elements in $\mathbb{F}_2$.

In the following, we first detail the changes to the input gates compared to a non-SIMD evaluation (§4.1). Then, we introduce new data management gates that are needed for re-organizing the elements on the wires to build circuits with high data-dependency (§4.2). Finally, we outline miscellaneous functions that are changed or added compared to a non-SIMD evaluation (§4.3).

Remark: Throughout this chapter, as well as in many ABY implementations, the value `nvals` indicates if a share is non-SIMD (i.e., `nvals` $= 1$) or SIMD (i.e., `nvals` > 1).

4.1 Changes to Input Gates

The ABY framework abstracts from the SIMD programming style and hence all function gates from §3 can be used exactly as for non-SIMD gates without requiring the developer to explicitly keep track of the dimensions. The only gates that explicitly require the developer to specify the dimension of SIMD gates are input gates §4.1.1, constant gates §4.1.2, and assert gates §4.1.3. Note, however, that the dimensions of shares that are input to a gate have to match or else an assertion error will occur.

唯一明确要求开发人员指定SIMD门的尺寸的门是输入门§4.1.1，常数门§4.1.2和断言门§4.1.3。但是，请注意，输入到门的共享的维度必须匹配，否则将发生断言错误

4.1.1 PutSIMDINGate

To create SIMD circuits, the `PutINGate` (see §3.1.1) is replaced by a `PutSIMDINGate` that additionally allows to specify the number of elements on a wire to be shared using the variable `nvals`. The method returns a `share` object that encapsulates the shared secrets and which can be used analogue to a non-SIMD `share`. Note that a call to `PutSIMDINGate` with `nvals`$=1$ is the same as a call to `PutINGate`.

```
share* PutSIMDINGate(uint32_t nvals, uint64_t val, uint32_t bitlen
    , e_role role);
share* PutSIMDINGate(uint32_t nvals, uint32_t val, uint32_t bitlen
    , e_role role);
```

```
share* PutSIMDINGate(uint32_t nvals, uint16_t val, uint32_t bitlen
    , e_role role);
share* PutSIMDINGate(uint32_t nvals,  uint8_t val, uint32_t bitlen
    , e_role role);

share* PutSIMDINGate(uint32_t nvals, uint64_t* val, uint32_t
    bitlen, e_role role);
share* PutSIMDINGate(uint32_t nvals, uint32_t* val, uint32_t
    bitlen, e_role role);
share* PutSIMDINGate(uint32_t nvals, uint16_t* val, uint32_t
    bitlen, e_role role);
share* PutSIMDINGate(uint32_t nvals,  uint8_t* val, uint32_t
    bitlen, e_role role);
```

Parameters

- nvals indicates the number of SIMD elements to be stored on a single wire.
- val is the input value loaded by one of the parties to generate the shared secret. This variable can be of type uint8_t, uint16_t, uint32_t, uint64_t or a pointer to an array of such a datatype. We give more information about the input format in the example below.
- bitlen states how many bits of the plaintext should be read and specifies the number of wires in the generated share.
- role defines the party which generates the share based on the input value provided. The role value can be CLIENT or SERVER.

PutSIMDINGate用于生成两个位长为4的共享，即money_bitlen = 4-bit，每个共享存储nvals = 3个元素。

Example In the example given below, PutSIMDINGate is used to generate two shares with bitlength 4, i.e., money_bitlen = 4-bit, and where each share stores nvals = 3 elements.

```
//Share creation with nvals as 3
uint32_t nvals = 3;
uint32_t alice_money[nvals] = {1, 3, 14};
uint32_t bob_money[nvals] = {6, 12, 5};
uint32_t money_bitlen = 4;

//Build input gates to obtain the corresponding SIMD shares
share* s_alice_money = circ->PutSIMDINGate(nvals, alice_money,
    money_bitlen, CLIENT);
share* s_bob_money = circ->PutSIMDINGate(nvals, bob_money,
    money_bitlen, SERVER);
```

Remark: The input format is such that all nvals values are written into separate array positions and the remaining unused bits are ignored. The following code shows a more complex input format example where two 10-bit values are secret-shared using the uint8_t type, that holds at most 8 bit values:

```
uint32_t nvals = 2;
uint32_t bitlen = 10;
uint32_t bytelen = (bitlen+(bitlen-1))/8; //2 bytes for 10 bits
uint8_t* vals[nvals * bytelen]; //uint8_t array with 2*2 positions

//First 10-bit value: 0x03FF
vals[0] = 0xFF;
vals[1] = 0x03;

//Second 10-bit value: 0x02AA
vals[2] = 0xAA;
vals[3] = 0x02;

//Build input gates to obtain the corresponding SIMD shares
share* s_val = circ->PutSIMDINGate(nvals, vals, bitlen, CLIENT);
```

Remark: As for the non-SIMD PutINGate, each party has to create an input gate for *both* roles. However, only the input of each party for its own role will be set in the circuit while the input to the gate of the other party will be ignored. In a similar fashion, PutSharedSIMDINGate can be used to input pre-shared SIMD inputs (see §3.1.2 for more information on pre-shared inputs).

4.1.2 PutSIMDCONSGate

与输入门类似，putcongate方法(见§3.1.3)在添加参数nvals的SIMD设置中工作时
也被putsimdcongate方法所取代。与SIMD输入门相反，SIMD常数门只接受一个
bit -bit长度的单一值val作为输入，并创建一个具有nvals val副本的SIMD共享。

Similar to the input gates, the PutCONSGate method (see §3.1.3) is also replaced by the method PutSIMDCONSGate when working in the SIMD setting that adds the parameter nvals. In contrast to the SIMD input gates, the SIMD constant gates only take a single value val of bitlen-bit length as input and create a SIMD share with nvals copies of val.

```
share* PutSIMDCONSGate(uint32_t nvals, uint64_t val, uint32_t
    bitlen);
```

Example In the example below, the constant 42 is copied nvals=3 times.

```
uint32_t nvals = 3;
uint32_t bitlen = 6;
uint64_t constant = 42;

//creates a SIMD share with 3 copies of the constant 42
share* s_val = circ->PutSIMDCONSGate(nvals, constant, bitlen);
```

4.1.3 PutSIMDAssertGate

当验证SIMD共享时，PutAssertGate方法(参见§3.4.2)被PutSIMDAssertGate方法所取代，该方法添加了参数nvals。输入格式与PutSIMDINGate相同

When verifying SIMD shares, the PutAssertGate method (see §3.4.2) is replaced by the PutSIMDAssertGate method that adds the parameter nvals. The input format is the same as for PutSIMDINGate (see §4.1.1).

```
share* PutSIMDAssertGate(share* in, uint32_t nvals, uint64_t*
   assert_val, uint32_t bitlen);
share* PutSIMDAssertGate(share* in, uint32_t nvals, uint32_t*
   assert_val, uint32_t bitlen);
share* PutSIMDAssertGate(share* in, uint32_t nvals, uint16_t*
   assert_val, uint32_t bitlen);
share* PutSIMDAssertGate(share* in, uint32_t nvals,  uint8_t*
   assert_val, uint32_t bitlen);
```

4.2 Data Management Gates

To allow data management for SIMD shares, we introduce several new gate types. Note that none of these gates changes the actual value on a wire. Instead, these gates allow to form SIMD shares of a certain format which can then be evaluated using existing function gates.

4.2.1 PutCombinerGate

The *combiner* gate takes as input a non-SIMD share input containing σ wires and joins them to a SIMD share with a single wire with $\texttt{nvals} = \sigma$ values. The combiner gate is used to group together non-SIMD values before inputting them into a SIMD operation. Alternatively, the wires can be input as multiple shares objects ina and inb, where the input shares can already be SIMD shares. In this case, it combines the values on all wires, e.g., if two wires with 3 and 4 values are input into the combiner gate, the resulting wire will hold 7 values.

```
share* PutCombinerGate(share* input);
share* PutCombinerGate(share* ina, share* inb);
```

Example In the example, which is graphically illustrated in Fig. 4.2, a non-SIMD 4-bit share s_val is transformed into a SIMD share s_out with $\texttt{nvals} = 4$ single bit values.

```
uint8_t val = 1;
share* s_val = circ->PutINGate(val, 4, CLIENT);
share* s_out = circ->PutCombinerGate(s_val);
```

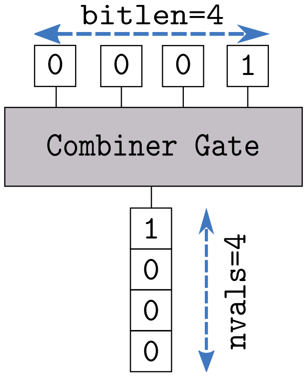


Figure 4.2: The combiner gate creates a SIMD share with nvals $= 4$ single bit values from a non-SIMD share with a $\sigma = 4$ bit element.

合并门从具有σ = 4位元素的非SIMD共享中创建具有nvals = 4位值的SIMD共享。

4.2.2 PutSplitterGate

The *splitter* gate takes as input a SIMD share input with a single wire with nvals values and splits it into a non-SIMD share with $\sigma =$ nvals wires, each with nvals $= 1$ values. The splitter gate is the reverse operation to the combiner gate and can be used to transform a SIMD gate back into a non-SIMD gate.

分路门将带有带有nvals值的单线的SIMD共享输入作为输入，并将其分割成带有σ = nvals值的单线的非SIMD共享，每条线的nvals值为1。分路门是组合门的反向操作，可用于将SIMD门转换回非SIMD门。

```
share* PutSplitterGate(share* input);
```

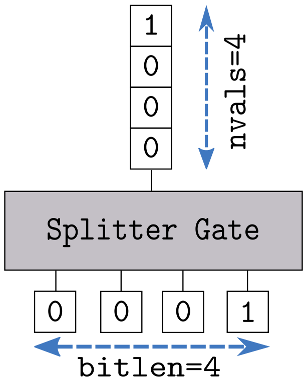


Figure 4.3: The splitter gate creates a non-SIMD share with $\sigma = 4$ wires from a SIMD share with a single wire and nvals $= 4$ elements.

Example In the example, which is graphically illustrated in Fig. 4.3, a SIMD single-bit share s_val with nvals $= 4$ is transformed into a non-SIMD share s_out with $\sigma = 4$ wires and nvals $= 1$.

```
uint8_t val[4] = {1, 0, 0, 0};
share* s_val = circ->PutSIMDINGate(4, val, 1, CLIENT);
```

```
share* s_out = circ->PutSplitterGate(s_val);
```

4.2.3 PutCombineAtPosGate

The *combine at position gate (CombineAtPosGate)* takes a SIMD share input with σ wires as input and combines the element at position pos on each wire of input into a new SIMD share with a single wire and σ values on that wire. The CombineAtPosGate is useful when a SIMD share needs to be transposed.

```
share* PutCombineAtPosGate(share* input, uint32_t pos);
```

Parameters

- input the input share object containing σ wires and nvals_in values on each wire, which requires joining.
- pos the position at which joining is performed. The value of pos must be in the range $\{0, \ldots, \texttt{nvals\_in}-1\}$.

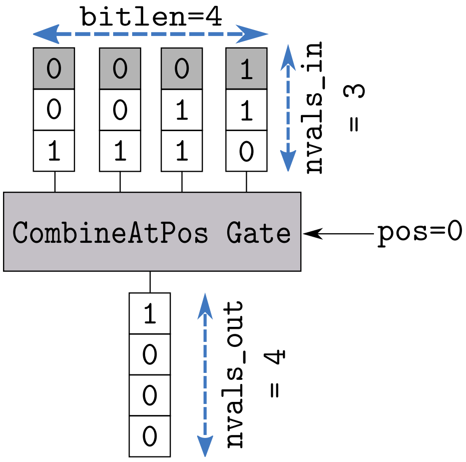


Figure 4.4: The CombineAtPos gate creates a SIMD share with a single wire that holds nvals_in = 4 values from a SIMD share with $\sigma = 4$ wires where each wire holds nvals = 3 values.

Example In the example, which is graphically illustrated in Fig. 4.4, a SIMD share s_val with $\sigma = 4$ wires and nvals = 3 values on each wire is transformed into a SIMD share s_out with $\sigma = 1$ wire which has nvals = 4 values.

```
uint8_t vals[3] = {1, 3, 14};

share* s_val = circ->PutSIMDINGate(3, vals, 4, SERVER);
share* s_out = circ->PutCombineAtPosGate(s_val, 0);
```

4.2.4 PutSubsetGate

The *subset* gate takes as input a SIMD share input with a single wire with multiple values and an arbitrary list of positions posids that is of size nvals_out. It returns a SIMD share with a single wire that consists of nvals_out values of input at the positions specified in posids.

```
share* PutSubsetGate(share* input, uint32_t* posids, uint32_t
    nvals_out);
```

Parameters

- input the input share with nvals_in > 1. If it contains more than one wire (σ>1), then the same subset of nvals is selected from every wire.
- posids an array of nvals_out positions to be selected from the input share. Every position must be in the range $\{0, \ldots,$ nvals_in$-1\}$. The number of supplied positions can be chosen freely. Positions can occur an arbitrary number of times.
- nvals_out the number of positions in posids and the number of values of the resulting output share.

Example In the example, which is graphically illustrated in Fig. 4.5, a SIMD share s_out with $\sigma = 2$ wires and nvals_out $= 3$ values is created from the positions $\{0, 2, 2\}$ of a share s_val with $\sigma = 2$ wires and nvals_in $= 4$ values.

```
uint32_t nvals_in = 4;
uint32_t nvals_out = 3;
uint8_t val[4] = {2, 1, 1, 3};
uint32_t posids[nvals_out] = {0, 2, 2};

share* s_val = circ->PutSIMDINGate(nvals_in, val, 2, SERVER);
share* s_out = circ->PutSubsetGate(s_val, posids, nvals_out);
```

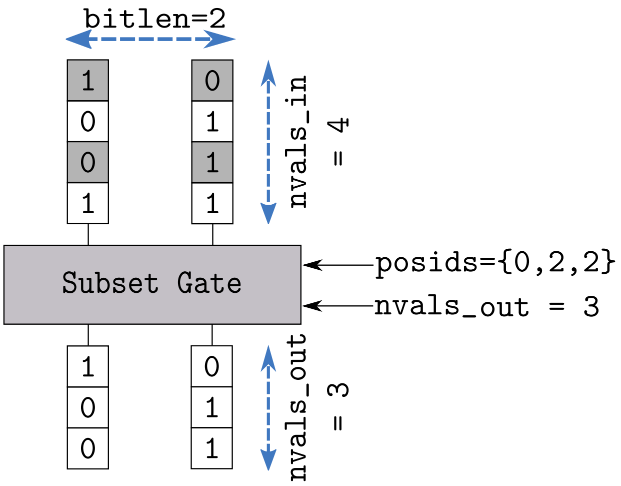


Figure 4.5: The subset gate creates a SIMD `share` with $\sigma = 2$ wires that hold `nvals` = 3 values each from a SIMD `share` with $\sigma = 2$ wires that hold `nvals_in` = 4 values each.

4.2.5 PutPermutationGate

The *permutation* gate takes as input a SIMD share `input` with σ wires each with `nvals_in` values and a list of positions `posids` with σ entries. It returns a single wire SIMD share `s_out` with `nvals=`σ values, where the i-th value of `s_out` comes from the i-th wire of `input` at position `posids[i]`.

```
share* PutPermutationGate(share* input, uint32_t* posids);
```

Parameters

- `input` the share with the σ input wires and `nvals_in` values from which the values should be taken.
- `posids` contains the position on the wires from `input` from which the values should be read. `posids` must have σ entries, i.e., one entry for each wire in the input share (its bitlength). Each position must be in the range $\{0, \dots,$ `nvals_in`$-1\}$ for the given wire.

Example In the example below that is depicted in Fig. 4.6, the share `s_out` with a single wire is assigned `nvals` = 4 values from the share `s_vals` with $\sigma = 4$ wires and `nvals_in` = 3 values. More detailed, after the `PutPermutationGate`, `s_out` contains the values on positions $\{0-0, 1-0, 2-2, 3-1\}$, where the notation $i-j$ denotes the j-th element on wire i of `s_vals`.

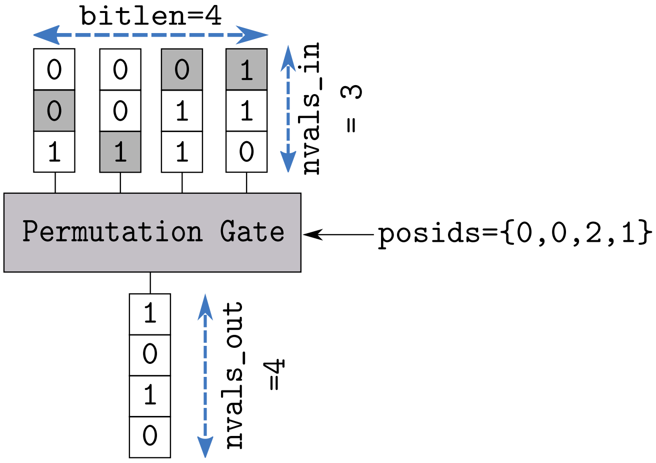


Figure 4.6: The permutation gate creates a SIMD `share` with a single wire that holds `nvals=4` values from a SIMD `share` with $\sigma = 4$ wires that each hold `nvals_in` $= 3$ values.

```
uint32_t bitlen = 4;
uint32_t nvals_in = 3;
uint8_t val[nvals_in] = {1, 3, 14};
uint32_t posids[bitlen] = {0, 0, 2, 1};

share *s_val, *s_out;
s_val = circ->PutSIMDINGate(nvals_in, vals, bitlen, SERVER);
s_out = circ->PutPermutationGate(s_vals, posids);
```

4.2.6 PutRepeaterGate

The *repeater* gate takes as input a non-SIMD share `input` and a integer value `nvals` and outputs a SIMD share that contains the values of `input` `nvals` times.

```
share* PutRepeaterGate(uint32_t nvals, share* input);
```

Parameters

- `nvals` size of the output SIMD share.
- `input` non-SIMD input share from which the value is taken that is “repeated” `nvals` times on the output share.

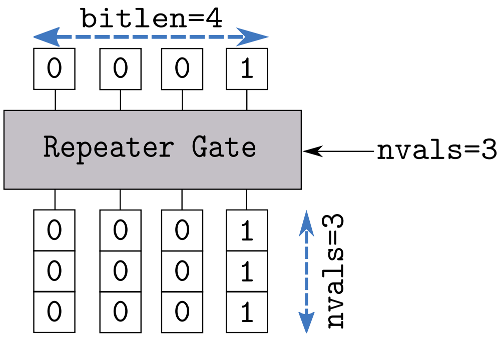


Figure 4.7: The repeater gate creates a SIMD share by copying the value of a non-SIMD input share nvals times.

Example In the example below, the SIMD share s_out which has wirelen = 4 wires and nvals = 3 values is created such that each element of s_out is equal to s_val.

```
uint32_t nvals=3;
uint32_t bitlen = 4;
uint8_t val = 1;
share* s_val = circ->PutINGate(val, bitlen, SERVER);
share* s_out = circ->PutRepeaterGate(nvals, s_val);
```

4.3 Misc Operations

The following operations are added to the share object to allow handling SIMD shares.

4.3.1 get_nvals_on_wire()

The get_nvals_on_wire() method can be used to obtain the nvals value on a specific wire with id wireid.

```
uint32_t get_nvals_on_wire(uint32_t wireid);
```

Parameters

- wireid value uniquely identifies a given wire in a share.

4.3.2 get_clear_value_vec()

The get_clear_value_vec() method is the SIMD pendant of the get_clear_value() (see §2.2.3) method and can be used to obtain the plaintext value of a share. In contrast to the get_clear_value() method, the get_clear_value_vec() uses the concept of "call by reference" and returns a vector of outputs, the bitlength of the outputs, and the number of output values of the share. The possible vector datatypes for the vector of outputs are uint32_t and uint64_t.

```
void get_clear_value_vec(uint32_t** vec, uint32_t* bitlen,
    uint32_t* nvals);
void get_clear_value_vec(uint64_t** vec, uint32_t* bitlen,
    uint32_t* nvals);
```

The first prototype is used for obtaining a uint32_t vector output, whereas second is used for obtaining a uint64_t vector output.

Parameters

- vec returns the vector which contains the output plain text information retrieved from the share object. This value is a reference and therefore modified by the method. The method is overloaded based on the datatype of vec.
- bitlen returns the bitlength of the values that are returned in vec.
- nvals returns the number of values of bitlength bitlen that are returned in vec.

Example The following example illustrates the use of get_clear_value_vec().

```
uint32_t bitlen = 8;
uint32_t nvals = 4;
uint8_t vals[nvals] = {64, 32, 128, 255};

uint32_t out_bitlen, out_nvals, *out_vals;

share* s_vals = circ->PutSIMDINGate(nvals, vals, bitlen, SERVER);
share* s_out = circ->PutOUTGate(s_vals, SERVER);

//as a result it holds that, out_vals[i] == vals[i], out_bitlen ==
    bitlen, and out_nvals == nvals
s_out->get_clear_value_vec(&out_vals, &out_bitlen, &out_nvals);
```

5 Benchmarking

ABY has several benchmarking routines built in. Set the desired macros in src/abycore/ABY_utils/ABYconstants.h:32 to 1 in order to print benchmarking numbers for each ABY execution. If you change these macros, you will need to recompile the core of ABY for the changes to have an effect. For this you need to run make cleanmore and then make again.

Listing 5.1: Benchmarking flags in src/abycore/ABY_utils/ABYconstants.h

```
32 #define PRINT_PERFORMANCE_STATS 1
33 #define PRINT_COMMUNICATION_STATS 1
34 #define BENCHONLINEPHASE 1
```

PRINT_PERFORMANCE_STATS will print gate counts and runtime information for both setup and online time. PRINT_COMMUNICATION_STATS shows communication numbers for sent and received data during several phases of the protocols. BENCHONLINEPHASE shows details runtime information of the online phase for every individual sharing type.

With all benchmarking flags set to 1 you will receive output that looks similar to the one shown in Listing 5.2.

Listing 5.2: Example Benchmarking Output

```
./min-euclidean-dist.exe -r 1 -n 1000 -d 2
Online time is distributed as follows:
Bool: local gates: 94.942, interactive gates: 47.419, layer finish
    : 23.019
Yao: local gates: 18.752, interactive gates: 6.147, layer finish:
    5.693
Yao Rev: local gates: 0.033, interactive gates: 0.025, layer
    finish: 0.024
Arith: local gates: 0.533, interactive gates: 0.753, layer finish:
     0.918
SPLUT: local gates: 0.039, interactive gates: 0.02, layer finish:
    0.179
Communication: 46.544
Complexities:
Boolean Sharing: ANDs: 202911 (1-bit) ; 999 (32-bit) ; Depth: 108
Total Vec AND: 203910
Total Non-Vec AND: 234879
XOR vals: 348874 gates: 286936
Comb gates: 0, CombStruct gates: 0, Perm gates: 0, Subset gates:
    0, Split gates: 0
Yao: ANDs: 31000 ; Depth: 6
Reverse Yao: ANDs: 0 ; Depth: 0
Arithmetic Sharing: MULs: 2000 ; Depth: 3
SP-LUT Sharing: OT-gates: Total OT gates = 0; Depth: 1
Total number of gates: 929800
```

```
Timings:
Total =     410.593 ms
Init =      0.184 ms
CircuitGen =    0.069 ms
Network =   0.607 ms
BaseOTs =   919.677 ms
Setup =     165.308 ms
OTExtension =   129.077 ms
Garbling =  35.68 ms
Online =    245.284 ms

Communication:
Total Sent / Rcv    5159400  bytes / 7181498 bytes
BaseOTs Sent / Rcv  149064  bytes / 149064 bytes
Setup Sent / Rcv    5083349  bytes / 5563333 bytes
OTExtension Sent / Rcv  5083349  bytes / 4571324 bytes
Garbling Sent / Rcv 0  bytes / 0 bytes
Online Sent / Rcv   76051  bytes / 1618165 bytes
```

The communication numbers are the Bytes that the individual party sent and received during the protocol execution. The *total time* is the time for the *setup phase* and the *online phase* combined. The time for the base-OTs is not counted into the total time, as this is a one-time expense that only happens when the connection between the two parties is established. The setup phase contains the times for OT-Extension and Garbling.

Bibliography

[DSZ15] D. Demmler, T. Schneider, and M. Zohner. ABY – a framework for efficient mixed-protocol secure two-party computation. In *Network and Distributed System Security (NDSS'15)*. The Internet Society, 2015. Code: `https://github.com/encryptogroup/ABY`.