

L1 - Faster Development and Benchmarking of Cryptographic Protocols

Axel Schröpfer, Florian Kerschbaum, Debmalya Biswas, Steffen Geißinger, and
Christoph Schütz

SAP Research Karlsruhe, Germany

{axel.schroepfer, florian.kerschbaum, debmalya.biswas,
steffen.geissinger, christoph.schuetz}@sap.com

Abstract. Secure Multi-party Computation (SMC) enables secure distributed computation of arbitrary functions of private inputs. Multiple techniques for SMC have been well studied and can be applied within cryptographic protocols, leading to large and complex protocols. Their implementation is difficult for an average programmer to understand, time consuming and potentially prone to errors. We introduce a new programming language dedicated to cryptographic protocols, which speeds up their implementation, the deployment of the running software, and furthermore provides integrated support for benchmarking.

1 Introduction

In many situations, especially in business contexts, multiple parties would like to compute a common function of their private inputs. Suppose several business partners of a supply chain wish to create an optimal supply chain master plan as in [7]. The master plan recommends the amount of products to be manufactured and the routes and times of shipping, in order to end up with minimal total costs of the supply chain. For this computation it requires the partners to expose sensible business data (e.g., production costs and capacities) to a central planning unit, i.e., trust in her. Although their will be a benefit due to reduced costs, the business partners are reluctant to share their sensible data with others, since they are very risk averse. Sharing their business data leads to potential risks, e.g., a loss in bargaining power. Secure Multi-party Computation (SMC) can overcome this risk by providing a privacy preserving solution to the computation.

SMC allows a set of n players, $P = P_1, \dots, P_n$, to jointly compute an arbitrary function of their private inputs, $f(x_1, \dots, x_n)$. The computation is privacy preserving, i.e. nothing else is revealed to a player than what is inferable by his private input and the outcome of the function. A cryptographic protocol is then run between the players in order to carry out the computation. Even if there are adversarial players, the constraints on correctness and privacy can be proven to hold under well stated settings. These settings consider the type of adversary as well as his computing power which can be bounded or unbounded. An adversary can be passive, i.e. following the protocol correctly but trying to learn

more or he can be active, by arbitrarily deviating. For the two-party case it has been proven by Yao in [12], that any arbitrary function is computable in privacy preserving fashion, using garbled binary circuits. This approach has been extended to the multi-party case in [1, 5]. The first multi-party approach [1] builds on secret sharing schemes. A player's secret s is split into m shares which are then distributed to m players. Players can compute intermediate results on the shares, and in the end a reconstruction is performed in order to receive the final result. Another multi-party approach [3] utilizes semantically secure, threshold, homomorphic encryption (HE) [4], a threshold public-key encryption scheme, where $E(x) \cdot E(y) = E(x + y)$ and x cannot be deduced by $E(x)$.

Although the theory behind SMC has been studied for more than two decades, practical implementations are rare. One reason we see for that is the lacking support by development tools. Beside the approach to start implementing the protocol directly in a programming language like Java, one may take advantage of one of the few existing SMC frameworks. The existing frameworks support the implementation of cryptographic protocols based on a particular SMC technique. *Fairplay* [6] for instance represents a two-party solution to the garbled circuit approach of [12]. It consists of two components. First, there is component which provides a compilation from code written in the high level programming language SFDL (secure function definition language) to a optimized Boolean circuit, described in SHDL (secure hardware definition language). Second, there is component which executes the binary circuit. A multi-party version is also available [2]. Another Framework is the *Virtual Ideal Functionality Framework* (VIFF) [11]. It represents a software system based on secret sharing and furthermore supports Shamir secret sharing and pseudo random secret sharing (PRSS). The system is implemented in Python, thus the code gets interpreted rather than compiled. An advantage of VIFF is its architecture principle, asynchronous parallel execution. Also based on secret sharing, *sharemind* [9] offers a framework for implementing cryptographic protocols. The private data in this implementation is shared over 32 bit integers and each protocol is carried out between a set of dedicated servers. What the frameworks have in common is, that they all base on a particular SMC technique. The frameworks also provide an abstraction from the cryptographic protocol, i.e., provide a domain specific language or predefined code (e.g., modules) to formulate the computation problem which then is translated into the cryptographic counterpart.

We provide a programming language, *L1*, dedicated to cryptographic protocols with support for various SMC techniques, namely so far garbled circuits, secret sharing and homomorphic encryption. Thus, our language allows also implementing cryptographic protocols which utilize more than just one SMC technique. Our language has built-in support for asynchronous message transmission between different instances (i.e., parties of the cryptographic protocol). Furthermore, it offers parallel execution of code. The syntax (resp., grammar) of our language is tailored to the needs of expressing cryptographic protocols. It aims to abstract from surrounding issues like network programming and thread synchronization. The simplicity of the language supports the programmer to

concentrate on the cryptographic protocol rather than technical issues and thus significantly speeds up protocol implementation. With its support for modules this is especially true for protocols composed from sub-protocols. The language also provides built-in support for benchmarking of execution time, number of messages and transmitted bytes. Through this feature, it is possible to benchmark protocols against other protocols in simulations or real world environments.

In the following sections we want to introduce the design of the new language, as well as that of the corresponding compiler. We furthermore highlight the configuration, deployment and benchmarking of the resulting distributed system. A section on open questions and a discussion conclude this document.

2 Syntax

The language has been tailored to what is necessary to implement a cryptographic protocol. Listing 1.1 depicts a sample of L1 code. The syntax of the language is motivated by that of *C*. The following listing gives a brief first overview which will be described in more detail.

```

1 //modules
2 include "key.l1";
3
4 //self defined functions
5 int newHash(int value) {
6     int hash;
7     ...
8     return hash;
9 }
10
11 //variables
12 int hash;
13
14 //assignment with expression
15 int salt = rand(1000) + 1;
16
17 //player dependent statement
18 1: {
19     //function call
20     hash = newHash(salt);
21     //message sending (non-blocking)
22     send(2, hash, "hash_value_from" + id());
23 }
24
25 //message receiving (blocking)
26 2: readInt("hash_value_from_1");
27
28 if (hash%2 == 0)
29     output("odd hash: " + hash);
30
31 //for-loop
32 for (int8 i = 0; i < 200; i = i + 1) <
33     //parallel execution
34     ...
35 >
36
37 //while-loop
38 while (condition) {
39     ...
40 }

```

Listing 1.1. L1 sample

In line 12 a variable of data type *int* with name *hash* is defined. The language supports the data types *string*, *bool*, *int8*, *int16*, *int32*, *int*, *prvk* and *pubk*. While *int* is unlimited in its size, the other integer types are restricted to the number of bits included in their names. *prvk* and *pubk* denote special types for private key and public key. Assignments and expressions are straightforward (as in *C*), depicted in line 15. The same is true for loops, presented at line 32 (for-loop) and line 38 (while-loop). Also the definition of functions from line 5 is straightforward, as well as the conditional execution of line 28. Line 2 shows how code from external files can be included as a module.

Also *C* style is the handling of code blocks with one distinction: a block of statements in curly braces will be executed sequentially, while a block of statements in angle brackets will be passed to a separate thread (i.e., being executed in parallel). If there are multiple parallel blocks, the next statement in line will be executed right after all threads of the group of parallel blocks terminated, i.e., the language provides a built-in barrier synchronization.

The language also provides code execution based on the identifier of the instance (i.e., a party of the cryptographic protocol). Line 18 and Line 26 show how identifier based code execution is expressed - by the ID of the instance, followed by a colon. The assignment of IDs to instances will be discussed in a later section.

The code in line 22 shows how a player can access his current ID, by calling the built-in function *id()*. The language provides a variety built-in functions, like cryptography routines (e.g., *sha1*), infrastructure routines (e.g., *send*) and other utility functions.

In summary, we designed the syntax (resp., grammar) with the goal of abstracting from everything except the cryptographic operations, such that the programmer can quickly concentrate on implementing the core functionality of the protocol. In the appendix we present three examples of protocols implementing secure multiplication, based on the different SMC techniques of Shamir Secret Sharing, Yao Garbled Circuits and Paillier Homomorphic Encryption.

3 Compiler

We build the compiler with *JavaCC*. The compiler takes as input a *L1*-source code file. It generates *Java*-source code as a class *PlayerDefaultProtocol*. For each ID which has been used in the *L1* source code, a separate class is generated, *Player{ID}Protocol*. We can then run the translated bytecode, e.g. from a script. In protocols, particular functionality is used regularly. We decided to externalize to libraries various functions which either can be optimized natively in *Java* or might not be implemented in *L1* at all. We make this libraries available to *L1* by built-in functions. For all calls to built-in functions contained in the *L1* source code, the generated *Java* source code includes calls to the library functions. For structural reasons, we split the functions into two packages: one for the infrastructure (i.e., message exchange) and one for cryptography tools. Figure

1 depicts the process from L1 source code to the executable program (i.e., Java Virtual Machine bytecode).

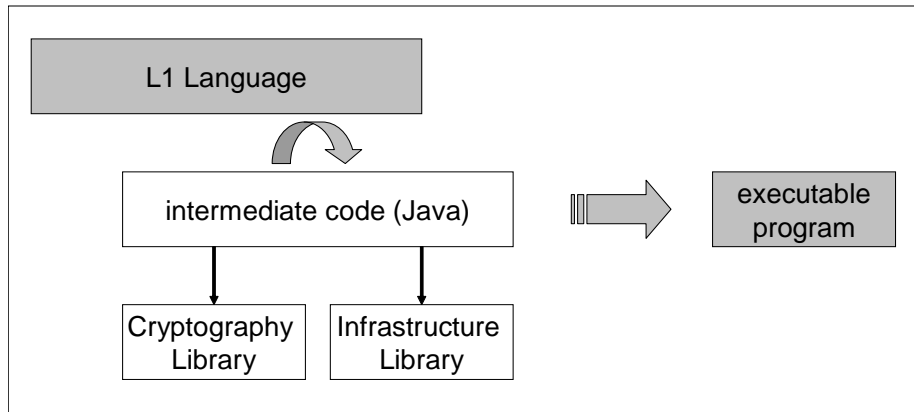


Fig. 1. L1 compiling

We build on an asynchronous messaging subsystem. We anticipate that an asynchronous approach will perform best. It not only supports simulating synchronous transmission, it also speeds up the computation in case the player of a protocol can directly continue with local computation after having triggered message sending. We provide a message layer which can be access by the Java code generated from the L1 program.

The message layer creates during instantiation a message server for receiving messages from other parties. Our messages have the format $messageId = value$. The messages server stores in his message buffer newly received messages (with according messageId and value). The message buffer can be accessed via the message layer by the built-in function $read(messageId)$ (optionally with datatype cast). This method will block the further execution of the cryptographic protocol until either the message with the given messageId has arrived or a read timeout occurs (also reading with no timeout is supported). When a message has been read, it is removed from the buffer. If a new message arrives with a messageId already present in the message buffer, the new message overwrites the one already in the buffer. The built-in function to $send(ID, messageId, value)$ will be translated into an asynchronous call of the message layer for transmitting messages in the above mentioned format. The message with the given messageId and value will be reliably transmitted to the protocol player with the given ID. A look-up (discussed in the next section) will provide the needed network addressing information. Due to the asynchronous call, the execution of the cryptographic protocol will be continued immediately and no answer except a receipt acknowledgement is expected by the message layer.

The library for cryptographic tools contains utility functions from cryptography which are used regularly. Alternatively, functions may be implemented directly in L1 rather than in the library. We built the most common functions into the library. In particular, the functions cover hashing, encryption (homomorphic), Shamir sharing, oblivious transfer and circuit garbling and evaluation. The library also covers file reading and writing and random number generation.

4 Configuration and Deployment

The generated Java code requires various configuration parameters in order to function properly. This is especially true for the TCP network addressing, i.e., connecting message servers in terms of hostnames (resp., IP addresses) and port numbers with IDs of the cryptographic protocol. Also other properties are valuable, e.g., the read message timeout (the time the message layer blocks while trying to get a message from the message buffer of the server, before aborting the protocol run). Determining the verbosity level of debug messages is another, commonly required parameter.

We therefore provide in our library a Java class *Settings* in order to be able to access persistent settings from an XML file. The settings may be utilized twice, by functions in the libraries and by calls to the built-in function *getParameter* (with cast support) in the L1 source code.

We provide next to the L1 compiler also a GUI-based configuration util *L1SettingsEditor* which supports quick setup of the execution environment. *L1SettingsEditor* furthermore provides a deployment feature. It will deploy all necessary files (library archives, settings file, as well as a start script) to a given directory (e.g., a share on the protocol party's host).

Figure 2 shows a screenshot of the settings editor.

5 Benchmarking

Benchmarking cryptographic protocols is one of the most relevant tasks for real world implementations of cryptographic protocols. In case of composed cryptographic protocols where individual components can be substituted, but have a variety of interdependencies, it is important to select the instances of sub-protocols which have the best performance under the parameters of the execution environment.

We provide in *L1* a built-in benchmarking capability. Benchmark recording can be done for arbitrary sections of the cryptographic protocol. A section is identified by a unique string. The built-in function *startBenchmark(name, parameterString)* starts recording benchmarks for the section *name* until *stopBenchmark(name)* is called. The parameter value is a comma separated string which specifies the items to benchmark. Currently accepted values are *time*, *byteCount* and *messageCount*. The parameter *benchmark_tree_filename* in the

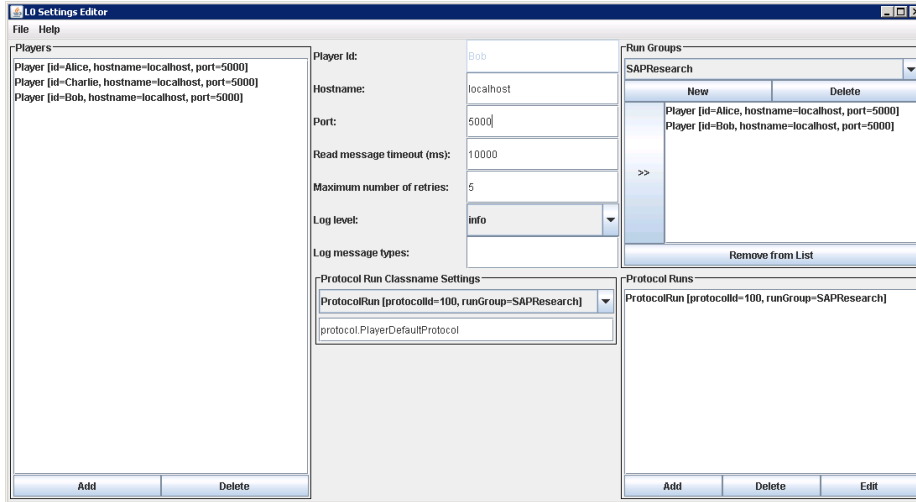


Fig. 2. L1 settings editor

settings file specifies the location of persistence as a serialized Java object. After the protocol run, one can easily access the results from a Java program for evaluation purposes (e.g., calculating statistics).

6 Open questions

With the L1 language and its compiler we aim at making the next step to bring theoretical work on SMC to practice. However, such a compiler raises new research questions which we have currently side stepped. An ongoing research direction in compiler design is the verification of the correctness of the compiler, i.e. verifying whether the properties of source and target code match. We have no proposal for addressing this in the current version, but have observed a more severe source of errors in practice. In many of our implementations of scientific papers the described protocols do not correctly implement the intended functionality. Thus the implementation in L1 is already incorrect and verifying whether the produced code matches the L1 code becomes rather pointless. Another ongoing research direction is the security of cryptographic protocols under various attacks, in particular side-channel attacks. Note that SMC protocols secure in the semi-honest model are side-channel free by design. One can lift this property and create more efficient protocols (e.g. done in [10]). We claim that if all primitives are implemented side-channel free, the composed SMC protocol will be side-channel free. We have not yet built an implementation that intends to be side-channel free. Our proposal for tackling both issues is to automatically generate a security proof – at least in the semi-honest model – from the L1 language. This proof would accompany the code and serve as proof that the protocol is secure. A verification at the L1 language layer improves over a verification at an

symbolic layer, since it can be directly translated into code. Given a valid security proof one then only needs to perform functional testing in order to identify incorrect protocols as in regular software engineering. In order to attest security against side-channel attacks, one only needs to verify the building blocks used in the compiler.

7 Discussion

Our new language *L1* offers the possibility to efficiently implement cryptographic protocols. Its various features speed up implementation time and also the process of evaluation and benchmarking.

We already have experienced the speed-up in practice, i.e., a real world example: we are currently implementing, based on the L1 language, a software system for secure supply chain management as part of the EU project SecureSCM [8]. We received very positive feedback with respect to implementation speed-up from the developers. Furthermore, we were able to quickly compare different sub-protocols. Through the benchmarking feature of the language, we could speed up the process of finding an optimal balance between the multiple security parameters of the cryptographic protocols and practical feasibility.

In addition, we point to several still open research questions for which we lack conclusive answers.

References

1. D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 503–513, 1990.
2. A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multiparty computation. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266, 2008.
3. R. Cramer, I. Damgard, and J. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Proceedings of EUROCRYPT, Lecture Notes in Computer Science 2045*, pages 280–299, 2001.
4. I. Damgard and M. Jurik. A generalisation, a simplification and some applications of pailliers probabilistic public-key system. In *Proceedings of International Conference on Theory and Practice of Public-Key Cryptography, Lecture Notes in Computer Science 1992*, pages 119–136, 2001.
5. O. Goldreich, S. Micali, , and A. Wigderson. How to play any mental game. In *Proceedings of the 19th Symposium on the Theory of Computing*, pages 218–229, 1987.
6. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - a secure two-party computation system. In *Proceedings of the USENIX security symposium*, pages 287–302, 2004.
7. A. Schröpfer, F. Kerschbaum, D. Sadkowiak, and R. Pibernik. Risk-aware secure supply chain master planning. In *In Proceedings of 7th International Workshop on Security in Information Systems*, pages 213–223, 2009.
8. SecureSCM. <http://www.securescm.org>.

9. Sharemind. <http://sharemind.cs.ut.ee/wiki/>.
10. T. Toft. *Primitives and Applications for Multi-party Computation*. PhD thesis, Department of Computer Science, University of Aarhus, 2007.
11. Virtual Ideal Functionality Framework. <http://www.viff.sk>.
12. A. Yao. How to generate and exchange secrets. In *In Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

Appendix A

```

1  int q = 733391; //common modulus
2  int8 players = 5;
3  int8 threshold = 2;
4
5  //1) let the dealer share a and b
6  1: {
7  int a = rand(100);
8  int aShares[0] = shamirShare(a, threshold, players, q);
9  for (int32 i = 1; i <= players; i = i + 1)
10     send(i, "aShare", aShares[i - 1]);
11 }
12 2: {
13 int b = rand(100);
14 int bShares[0] = shamirShare(b, threshold, players, q);
15 for (int32 i = 1; i <= players; i = i + 1)
16     send(i, "bShare", bShares[i - 1]);
17 }
18 int aShare = readInt("aShare");
19 int bShare = readInt("bShare");
20
21 //2) compute the product
22 int c = aShare * bShare;
23 int cShares[0] = shamirShare(c, threshold, players, q);
24 for (int32 i = 1; i <= players; i = i + 1)
25     send(i, "cShareFrom" + id(), cShares[i - 1]);
26
27 int cSharesFromOthers[players];
28 for (int32 i = 1; i <= players; i = i + 1)
29     cSharesFromOthers[i - 1] = readInt("cShareFrom" + i);
30
31 int abShare = vanDerMondeSum(cSharesFromOthers, q);
32
33
34 //3) continue with next operations (addition or multiplication)
35 //...
36
37 //4) reconstruct
38 send(3, "abShareFrom" + id(), abShare);
39 3: {
40 int abSharesFromOthers[players];
41 for (int32 i = 1; i <= players; i = i + 1)
42     abSharesFromOthers[i - 1] = readInt("abShareFrom" + i);
43
44 int ab = shamirReconstruct(abSharesFromOthers, q);
45 output("Output is a*b=" + ab + ".");
46 }

```

Listing 1.2. Multi-party multiplication based on Shamir Secret Sharing

```

1  int split_mul_p1(int x_1, int y_1, prvk paillierPrivateKey) {
2      int N = getModulus(paillierPrivateKey);
3      int N2 = N*N;
4      pubk paillierPublicKey = getPublicKey(paillierPrivateKey);
5      int x_1_enc = encrypt(paillierPublicKey, x_1);
6      int y_1_enc = encrypt(paillierPublicKey, y_1);
7      send(2, "x1Enc", x_1_enc);
8      send(2, "y1Enc", y_1_enc);
9      int v = readInt("v");
10     int v_dec = decrypt(paillierPrivateKey, v);
11     int z_1 = (v_dec + (x_1*y_1)) % N;
12     return z_1;
13 }
14
15 int split_mul_p2(int x_2, int y_2, pubk paillierPublicKey) {
16     int x_1_enc = readInt("x1Enc");
17     int y_1_enc = readInt("y1Enc");
18     int N = getModulus(paillierPublicKey);
19     int N2 = N*N;
20     int r = rand(N);
21     int v = (modPow(x_1_enc, y_2, N2)*modPow(y_1_enc, x_2, N2)*
22     encrypt(paillierPublicKey, -r)) % N2;
23     send(1, "v", v);
24     int z_2 = (r + x_2*y_2) % N;
25     return z_2;
26 }
27
28 1: {
29 //initilize keys
30 prvk paillierPrivateKey = createPaillierPrivateKey(1024);
31 int modulus = getModulus(paillierPrivateKey);
32 pubk paillierPublicKey = getPublicKey(paillierPrivateKey);
33 send(2, "pubkString", serializeKey(paillierPublicKey));
34
35 //inputs
36 int x_1 = 2;
37 int y_1 = 3;
38
39 //run protocol core
40 int z_1 = split_mul_p1(x_1, y_1, paillierPrivateKey);
41 int z_2 = readInt("z_2");
42 output(((z_1+z_2)%modulus) == 48);
43 }
44
45 2: {
46 //initilize keys
47 string pubkString = readStringMaxTimeout("pubkString");
48 pubk paillierPublicKey = deserializePaillierPublicKey(pubkString);
49 int modulus = getModulus(paillierPublicKey);
50
51 //inputs
52 int x_2 = 4;
53 int y_2 = 5;
54
55 //run protocol core
56 int z_2 = split_mul_p2(x_2, y_2, paillierPublicKey);
57 send(1, "z_2", z_2);
58 }

```

Listing 1.3. Two-party multiplication based on Paillier Homomorphic Encryption

```

1  int32 bits = 32;
2
3  1: {
4  //Bob is the encrypter
5
6  //read binary circuit from file
7  int32 gates[0][0] = loadMatrixInt32("mul32.gates");
8  int8 tt[0][0] = loadMatrixInt8("mul32.tt");
9  int32 resultPins[0] = loadMatrixInt32("mul32.result")[0];
10 int32 numberOfWires = resultPins[maxIndex(resultPins)] + 2 * bits;
11 send(2, "numberOfWires", numberOfWires);
12
13 //Bob's input
14 int32 bob = 8;
15 int32 input[bits];
16 int8 bobBits[0] = binary(bob, bits, true);
17 for (int32 i = 0; i < bits; i = i + 1)
18     input[i] = bobBits[i];
19
20 //create keys for all wires
21 int wireCodes[0][0] = yao2pGetWireCodes(numberOfWires);
22 int wires[numberOfWires];
23
24 //encode Bob's input and send it to Alice
25 for (int32 i = 0; i < length(input); i = i + 1)
26     wires[i] = wireCodes[i][input[i]];
27 send(2, "wires", wires);
28
29 //perform OT for each of Alice's input bits
30 for(int32 j = 0; j < bits; j = j+1) {
31     prvK otPrivateKey = createOTPrivateKey(2, 160);
32     pubK otPublicKey = getPublicKey(otPrivateKey);
33     send(2, "otPublicKey"+j, serializeKey(otPublicKey));
34     int encodedOTChoice = readInt("encodedOTChoice"+j);
35     int cipherRandom = randBits(32);
36     int ciphers[0] = createOTCiphers(otPrivateKey, wireCodes[bits+j],
37                                     cipherRandom, encodedOTChoice);
38     send(2, "cipherRandom"+j, cipherRandom);
39     for (int32 i = 0; i < 2; i = i + 1)
40         send(2, "ciphers" + i+"_"+j, ciphers[i]);
41 }
42
43 //garble, permit and encrypt the circuit
44 int pegtt[0][0] = yao2pGetPermutedEncryptedGarbledTruthTable(gates,
45                                                                tt, wireCodes);
46 int outputTranslationTable[0][0] = yao2pGetOutputTranslationTable(
47     gates, wireCodes, resultPins);
48
49 //transfer the secured circuit and the output mapping to Alice
50 send(2, "pegtt", pegtt);
51 send(2, "ott", outputTranslationTable);
52
53 //read output from Alice after she has evaluated
54 int32 output[0][0] = readMatrixInt32("output");
55 }
56
57 2: {
58 //Alice is the evaluator
59
60 //initialize circuit
61 int32 gates[0][0] = loadMatrixInt32("mul32.gates");
62 int32 numberOfWires = readInt32("numberOfWires");
63 int wires[0] = readArrayInt("wires");
64
65 //Alice's input
66 int32 alice = 36;
67 int8 aliceBits[0] = binary(alice, bits, true);

```

```

68
69 //perform OT for each of Alice's input bits
70 for(int32 i = 0; i < bits; i = i+1) {
71     string keyString = readString("otPublicKey"+i);
72     pubk otPublicKey = deserializeOTPublicKey(keyString);
73     int pad = randBits(160);
74     int8 choice = aliceBits[i];
75     int encodedOTChoice = encodeOTChoice(otPublicKey, choice, pad);
76     send(1, "encodedOTChoice"+i, encodedOTChoice);
77     int cipherRandom = readInt("cipherRandom"+i);
78     int ciphers[2];
79
80     for (int32 j = 0; j < 2; j = j + 1)
81         ciphers[j] = readInt("ciphers" + j+"_" +i);
82
83     wires[bits + i] = decryptOTCipher(otPublicKey, ciphers,
84                                     cipherRandom, choice, pad);
85 }
86
87 //receive secured circuit and the otuput mapping from Bob
88 int pegtt[0][0] = readMatrixInt("pegtt");
89 int ott[0][0] = readMatrixInt("ott");
90
91 //evaluate circuit
92 int32 output[0][0] = yao2pEvaluateCircuit(gates, pegtt, wires, ott);
93
94 //output result
95 for (int32 i = 0; i < length(output); i = i + 1)
96     output("wire_id:_" + output[i][0] + "_value:_" + output[i][1]);
97
98 //transmit output to Bob
99 send(1, "output", output);
100 }

```

Listing 1.4. Two-party multiplication based on Yao Garbled Circuits