L1 - A Programming Language for Mixed-Protocol Secure Computation

Axel Schröpfer[1], Florian Kerschbaum[1], and Günter Müller[2]

[1] SAP Research
{axel.schroepfer,florian.kerschbaum}@sap.com,
[2] Universität Freiburg
guenter.mueller@iig.uni-freiburg.de,

Abstract. Secure Computation (SC) enables secure distributed computation of arbitrary functions of private inputs. It has many useful applications, e.g. benchmarking or auctions. Several general protocols for SC have been proposed and recently been implemented in a number of compilers and frameworks. These compilers or frameworks implement one general SC protocol and then require the programmer to implement the function he wants the protocol to compute. Performance remains a challenge for this approach and it has been realized early on that special protocols for important problems can deliver superior performance. In this paper we propose a new programming language (L1) which enables efficient implementation of special protocols potentially mixing several general SC protocols. We show with two case studies – one for computation of the median and one for weighted average – that special protocols and mixed-protocol implementations in our programming language L1 can lead to superior performance.

1 Introduction

Secure Computation (SC) allows a set of n players P_i to jointly compute an arbitrary function $f()$ of their private inputs x_i (i.e. P_i has x_i): $f(x_1, \ldots, x_n)$. The computation is privacy-preserving, i.e. it reveals nothing to a player except what can be inferred by his private input and the output of the function. It has many useful applications, e.g. benchmarking [1, 2] and auctions [3].

Even if there are adversarial players, the guarantees for correctness and privacy can be proven to hold in well stated models. These models consider the type of adversary, active (malicious) or passive (semi-honest), and his computing power, bounded or unbounded. A passive adversary follows the protocol as prescribed but tries to learn additional information, while an active adversary may arbitrarily deviate from the protocol.

We distinguish two-party and multi-party (more than two) SC. The first general protocol for two-party SC was presented in [4]. It also defined the approach underlying all general SC protocols. First, the function to be computed is translated into a circuit. Second, each gate of the circuit is implemented using

a secure protocol. Two types of gates – multiplication (logical and) and addition (exclusive-or) – suffice to compute any function. This approach reduces constructing a general SC protocol to two protocols implementing these two types of gates. For multi-party SC in the cryptographic model (computationally bounded adversary) with binary circuits this has been solved in [5]. For multi-party SC in the information-theoretic model (unbounded adversary) with arithmetic circuits this has been solved in [6]. Both protocols need to be run interactively, i.e. the number of rounds corresponds to the multiplicative depth of the circuit. Multi-party SC in the cryptographic model has been improved to a constant number of rounds in [7].

Another useful tool for SC is homomorphic encryption (HE), e.g. [8–10]. In HE one operation on the ciphertext maps to another operation on the plaintexts. Recently [9] it was shown that this can be performed for logical not-and gates which also suffice to compute any function. Threshold HE (for only addition) can be used to implement multi-party SC in the cryptographic model [11]. Also two-party SC can be implemented using HE by adapting the protocol from [12].

In a seminal paper – FairPlay [13] – Malkhi et al. build on the general approach for SC by building a compiler that translates the function specified in a programming language into a circuit. This circuit is then executed using a general SC protocol. In [13] the general protocol was [4]. Later others followed: [14] for [15], [16] for [6], and [17] for [7].

A critical aspect of SC remains performance. Even optimized protocols report a slowdown of several tens of thousand compared to non-privacy-preserving implementation [2]. It has been realized early on that special protocols for important problems can provide better performance [18]. Yet these protocols are not supported in existing SC compilers and frameworks.

This paper contributes

- a *programming language* (L1) which can implement special SC protocols. It currently supports general SC protocols [4, 6, 11] and HE [8, 10], but it is also easily extensible to other protocols and encryption schemes.
- a *compiler* that translates L1 into Java code.
- a *benchmarking framework* built into the language and compiler that supports measuring SC protocols.
- two *case studies*: one for computation of the median which supports that special SC protocols can be faster than general SC protocols and one for weighted average which supports that mixing general SC protocols can be faster than a single general SC protocol.

The remainder of the paper is structured as follows: Section 2 reviews related work, in particular other compilers and frameworks for SC. Section 3 describes the L1 language, compiler and benchmarking framework. We present the case studies in Section 4 and our conclusions in Section 5.

2 Related Work

2.1 FairPlay

FairPlay [13] was the first compiler implementing SC. It implements Yao's two-party SC [4]. It provides a simple programming language for specifying the function to be computed. There are some restrictions on this function, such as no variable number of iterations in loops and no recursion. The language provides arrays and variable indexing into arrays which is reported as one of the most complex functions [4]. The compiler translates this language into a circuit which is then executed in Yao's protocol.

The language (SFDL) introduces some of the basic concepts for languages for SMC, such as input and output definition for variables.

2.2 FairPlayMP

FairPlayMP extends FairPlay to the multi-party setting. It is based on the protocol of [7] which is also an extension of [4] to the multi-party setting. Some code, e.g. in the compiler of the language to the circuit can therefore be shared. The interpretation of the circuit has to completely change, of course. FairPlayMP includes a novel preprocessing phase in order to reduce the communication overhead.

It also implements a popular concept in multi-party SC. Not all parties need to participate equally in the computation. Instead one can separate input, computation and output nodes. As long as there are sufficiently many computation nodes (usually at least three) the computation remains secure, but communication cost has been significantly reduced.

Its programming language includes some necessary extension for multi-party problems and some additional operators.

2.3 Virtual Ideal Functionality Framework

VIFF [14] is the basis for the first commercial application of SC [3]. It implements multi-party SC in the information-theoretic model [15]. It does not yet provide a high-level programming language for the functionality. Instead it extends the Python language to use operators on the basic data types. It implements the basic protocols for [15] as objects.

A high-level language (SMCL) for VIFF has been specified in [19]. It extends FairPlay's language by some features such as loops with public number of iterations and allows security analysis similar to information flow analysis. For this purpose it tags all variables in a lattice as either secret, private or public.

2.4 Sharemind

Sharemind [16] is a framework built for experimenting with privacy-preserving data mining. It implements the SC protocol from [6]. It has been optimized for

speed of simple operations, such as vector products. It therefore offers vectorized operations.

It is currently being extended with a programming language (SecreC) for specifying the function to be computed.

3 L1 Language

The L1 language was designed to program special SC protocols potentially mixing techniques from multiple general SC protocols. We expect this mix to enable significantly more efficient SC protocols as shown in our case studies in Section 4.

We have built a compiler that translates a SC protocol encoded in L1 into a set of Java programs – one for each player in the protocol. The players can then compile and execute the Java program in order to run the protocol programmed in L1.

3.1 Syntax

We chose our grammar similar to the familiar languages of Java and C. The L1 language contains the following constructs: variables of different data types in different composites (scalar, 1-dimensional and 2-dimensional array), expressions, control flow (*if, while, for*, sequential and parallel execution, player-specific code), functions (user-defined and built-in) and modules.

Listing 1.1 shows a sample of L1 source code demonstrating all constructs of the language. Variable types have straight-forward definitions. Variables of type *bool* can have values *true* or *false*, *string* variables can contain strings. Variables of type *int32* can be 32-bit signed integer values while those of type *int* can contain arbitrarily long integer values (e.g. SC shares). Finally, *prvk* and *pubk* variables carry public and private keys. As in almost all other imperative programming languages, expressions in L1 consist of operands which may be connected through operators. Operands can be function calls, variables or constants. The basic control flow constructs in L1 are quite standard for an imperative programming language and closely adhere to Java. L1 provides the basic constructs *if-else*, *for* and *while* in combination with basic blocks. Another well-known feature which is contained in L1 is code modularization, i.e., code can be separated in multiple files which then are included.

3.2 Parallel Execution

Besides the Java semantics of sequential statements and basic blocks L1 offers parallel execution of basic blocks. As mentioned before SC protocols can be very computational intensive [2] challenging the performance of a single CPU. In some cases [3, 20] it is already known that SC protocols can be quite efficiently parallelized capitalizing on the trend to multi-core CPUs. Since most large-scale SC problems have not yet been tackled due to performance concerns and CPU speed does not seem to continue to increase exponentially, we expect that future implementations will need to heavily exploit parallelism.

```
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   startBenchmark("hash");
21   hash = newHash(salt);
22   stopBenchmark("hash");
23
24   //message sending (non-blocking)
25   send(2,hash,"hash_value_from"+id());
26 }
27
28 //message receiving (blocking)
29 2: readInt("hash_value_from_1");
30
31 if (hash%2 == 0)
32   output("odd hash: "+hash);
33
34 //for-loop
35 for (int32 i=0; i<200; i=i+1) <
36 //parallel execution
37   ...
38 >
39
40 //while-loop
41 while (condition) {
42   ...
43 }
```

Listing 1.1: L1 sample

L1 offers a unique feature for the definition of parallel code sections. Basic blocks to be execute as a new thread are specified by angle brackets as delimiters (instead of curly brackets). The compiler inserts the necessary instructions into the Java code in order to spawn and execute a new thread containing this basic block. Afterwards, the execution continues and any adjacent parallel basic blocks are spawned and run in parallel threads. L1 will also synchronize those threads before returning to sequential processing.

All parallel threads register with a barrier before executing the L1 basic block. A barrier is a synchronization mechanism that blocks execution until all

registered threads have finished. The compiler uses the Java standard library class for barriers. After spawning parallel threads the L1 compiler inserts a call to the barrier to wait until all threads have finished before executing the next sequential statement.

Listing 1.2 shows an example for thread synchronization in L1. Line 1 contains an initial sequential statement that outputs "S1". The body of the *for* loop is a parallel basic block which will spawn and run two parallel threads. One outputs "P1" and one "P2". The last statement is a sequential one again and outputs "S2". The barrier mechanism of L1 synchronizes the threads, such that the last statement will always be executed last, i.e. the first and last output of the program will always be "S1" and "S2", respectively. The only two possible traces of the program are: S1, P1, P2, S2 and S1, P2, P1, S2.

We show the code generated by the L1 compiler in Listing 1.3. First, a new barrier (class `RegistrationBarrier`) is created. Second, an instance of class `ParallelStep` is created in a for loop wrapping the parallel basic block. The barrier receives a call to method `registerThread` passing the instance of the parallel basic block. The barrier increments its counter and then returns to the `ParallelStep` instance calling its method `registeredAtBarrier` signaling successful completion of the registration. In this method a new thread for the parallel basic block is created and started which, after the basic block finished, calls the method `reachedBarrier` of the barrier decreasing its counter. The main

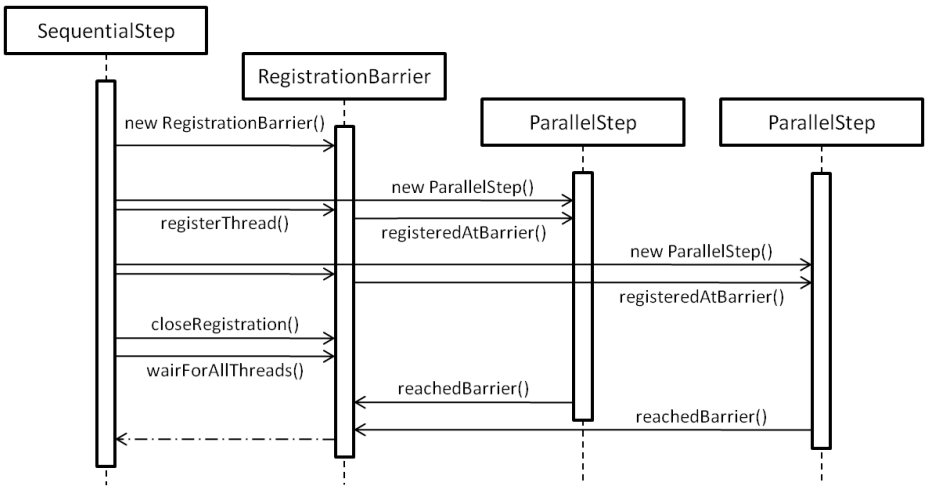


Fig. 1: Sequence Diagram for Parallel Execution

thread of the sequential program first calls method `closeRegistration` and then waits at the barrier using method `waitForAllThreads`. This method returns when the counter of the barrier is decrement to zero and the sequential program may continue. Figure 1 shows the sequence diagram for this interaction.

```
output("S1");
for (int32 i=1; i<=2; i=i+1)
<
  output("P"+i);
>
output("S2");
```

Listing 1.2: L1 Parallel Execution

```
public void step() {
  BuiltInFunctions.output("S1");
  RegistrationBarrier barrier =
    new RegistrationBarrier();
  for (i = 0;i < 2;i=i+1) {
    barrier.registerThread(
    new ParallelStep(this, stepThis) {
      public Integer sum;
      public Integer j;
      public Integer i;

      public void init() {
        sum = DefaultValue.newInteger();
        j = DefaultValue.newInteger();

        //copy instead of reference
        i = (Integer) parent.getClass().getField("i").get(
            parent);
      }

      public void step() {
        for (j = 1;j <= 2;j=j+1) {
          //body
          sum = sum + 1;
        }
        BuiltInFunctions.output("P" + i + ":" + sum);
      }
    });
  }
  barrier.closeRegistration();
  barrier.waitForAllThreads();

  BuiltInFunctions.output("S2");
}
```

Listing 1.3: L1 sample

3.3 Player-Specific Code

In many SC protocols, particularly in almost all general multi-party SC protocols, all players execute the same code (just on different data). The L1 compiler therefore produces several instances of the Java code – one for each player.

Some SC protocols, e.g. Yao's two-party SC [4], deviate from this pattern and execute different code for different players. Since the player identifier is accessible within L1, the differentiation could be performed at run-time using

an *if* statement. Instead we chose for performance reasons to differentiate at compile time and potentially produce different Java code for each player. A programmer can specify *player-specific code* sections in the L1 source. A player-specific code section is a statement or basic block prepended by the identifier of the player to execute this code followed by colon. Line 18 of Listing 1.1 shows an example of a player-specific code section.

The interpretation at compile time ensures leaner code at each player that only needs to execute the statements for this player. Furthermore we feel that it makes the L1 code easier to read and maintain.

3.4 Built-In Functions

Like procedural languages L1 structures its code into functions, but L1 offers two types of functions: user-defined and built-in functions. User-defined functions are specified and compiled as expected from the similarity to the Java language. As a constraint we currently require the definition of a user-defined function before its first invocation. Line 5 of Listing 1.1 shows an example of a function definition. Built-in functions are programmed in Java and not L1, but can be called from L1 just like any other function. Built-in functions are defined in a Java class `BuiltInFunctions` of the compiler. The compiler uses reflection in order to import the built-in functions. Using built-in functions the compiler can be extended with new features. Many features of the L1 language have been implemented as built-in functions. Two which are particularly worth mentioning are messaging and benchmarking.

Messaging
Messaging allows the transmission of messages between players enabling the distributed (secure) computation. L1 provides two sub-systems both based on TCP/IP: synchronous and asynchronous. The built-in functions *send* and *sendSync* send messages to other players (line 25). Their parameters are the identifier of the receiving player, a name for the message and its value. If the identifier of the player is 0, the player will broadcast to all other players. The name of the message is a replacement of its address and used by the recipient to retrieve the message in case of asynchronous communication.

The asynchronous *send* function implements non-blocking behavior (i.e., the next statement in line will be executed immediately). The synchronous *sendSync* will block the execution until the message has been acknowledged by all recipients. Synchronous send also supports an optional timeout parameter. Furthermore L1 also supports buffered sends which bundle several send invocations.

The recipient has a built-in receive (*read*) function for every data type. These functions require the message name as a parameter (line 29). Message receiving is always blocking, i.e., the read function will block and wait until the message with the specified name has been received. An optional timeout can be specified as a second parameter or else a default timeout is used.

Benchmarking
The design goal of L1 is programming faster SC protocols. Measuring the performance improvement therefore enables verifying whether this goal has been reached. L1 provides a benchmarking sub-system using built-in functions. Several benchmarks can be measured in parallel. Each benchmark is started by calling the built-in function *startBenchmark* (line 20) and stopped by calling *stopBenchmark* (line 22). Its parameter is the name for this benchmark called a benchmarking section. L1 implicitly takes care of multiple threads by internally appending the thread identifier to the name. In a benchmarking section the following quantities are captured: run time (wall clock time), number of messages sent or received, number of bytes sent or received. These correspond to computation and communication complexity in theoretic papers on SC.

4 Case Studies

Using two case studies we exemplify the performance benefit of using L1. In the first case study we compare two SC protocols for median computation, one entirely implemented using Yao's protocol [4] in FairPlay [13] and one implemented in L1. Although both implement the same function, the performance results are quite different. In the second case study we compare two SC protocols for weighted average computation, again one entirely implemented using Yao's protocol and one implemented in L1 also using an adaptation of Goethals et al.'s protocol [12].

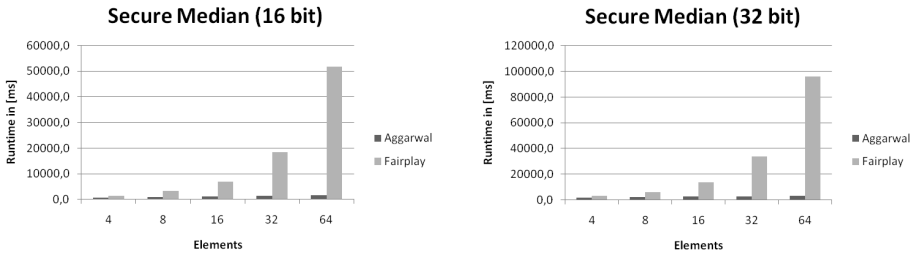
Each reported runtime measurement is the average of 20 samples. All tests were performed on two 2GHz Intel CoreDuo T7200 dual CPU machines with 2GB RAM each.

4.1 Median

Imagine two players, Alice and Bob, each with n elements drawn from a finite domain. They want to compute the median of their joint set of elements, i.e. the n-th ranked element in their combined, (ascendingly) sorted set, but do not want to disclose any of their other elements to the other party.

Aggarwal et al. proposed the following protocol for this problem [21]. Alice and Bob compare the median of their individual sets using a SC protocol for comparison, i.e. they only learn the result of the comparison, but not each other's input values. The party with the lower value selects the upper half of its elements and the party with the higher value selects the lower half of its elements. They then repeat the comparison with set half the initial size and continue doing so until the sets are of size 1.

We implement this algorithm in L1. We also perform comparisons in L1 using Yao's protocol, but do not show the code in the paper, since it is quite standard and import the function via a module. Note that using Yao's protocol one could implement the same algorithm, but unfortunately FairPlay does not support



(a) Secure Median with 16 bit

(b) Secure Median with 32 bit

Elements per Player	Aggarwal 16 bit	Fairplay 16 bit	Aggarwal 32 bit	Fairplay 32 bit
4	730,5	1525,9	1439,9	2838,4
8	931,3	3308,9	1885,3	6001,1
16	1164,9	7043,3	2353,2	13543
32	1395,4	18473,4	2793,9	33950,8
64	1620,4	51760,8	3250,2	96166,9

Table 1: Runtime in [ms] for Secure Median with 16 and 32 bit

the necessary operations, such as division (or shift). We therefore use FairPlay's example for median from its distribution.

The key insight of [21] is that the result of the comparison and subsequent selection of elements can be public (known to both parties), since it can be inferred from the (public) result of the computation. FairPlay's problem is that this insight cannot be implemented in its code while L1 can implement it. The variable indexing into the array of elements is an $O(n)$ operation in FairPlay while it is almost free in L1, since it does not require any communication.

We ran the protocols for varying numbers of elements (4, 8, 16, 32 and 64) and also for varying input bit lengths (16 and 32). Table 1 shows the runtime. Figures 2a and 2b depict the results as graphs. L1 always outperforms FairPlay and L1's advantage is increasing with an increasing number of elements. Unfortunately the results are distorted by FairPlay's lack of operations, but we believe that even if the more efficient algorithm would have been implemented in Yao's protocol L1's advantage of public selection of the remaining elements would have prevailed.

4.2 Weighted Average

Suppose Alice has n elements and Bob has m elements drawn from a finite domain. Let c be the sum of Alice's elements and d be the sum of Bob's. Furthermore, Alice and Bob share a weight w, such that Alice has w_A, Bob has w_B and $w = w_A + w_B$. This also covers less general problems where either only Alice or only Bob has the weight w. They want to jointly (and securely) compute the

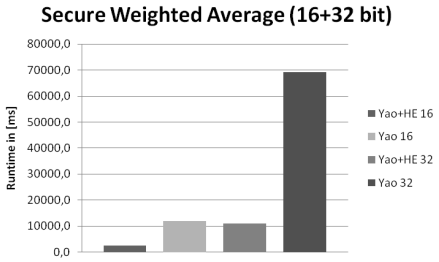


Fig. 2: Secure Weighted Average with 16 and 32 bit

Input Size	GC+HE	GC
16	2446,2	12034,1
32	11009,8	69149,6

Table 2: Runtime in [ms] for Secure Weighted Average with 16 and 32 bit

weighted average f:

$$f = \frac{(c+d)(w_A + w_B)}{n+m} \tag{1}$$

We implement this formula in Yao's protocol [4]. Due to a minor implementation error in FairPlay we had to resort to our own implementation of [4]. We then replaced the multiplication in the divisor in Yao's protocol by a variant of the protocol from [12]. We briefly review this variant.

Let Alice have a and Bob b they can compute $x + y = ab$ using HE. Alice sends $E_A(a)$ to Bob who computes $E_A(a)^b E_A(R) = E_A(ab + R)$ and returns it to Alice. Alice decrypts and stores the result as x while Bob sets $y = -R$. When combining Yao's protocol with HE care must be taken when choosing the length of the secret number R. Secret shares, such as in $w_A + w_B$ have significantly less bits than the key length of most HE schemes. Therefore R should be chosen longer according to the technique from [22] and later reduced using a modulo operation on the plaintexts.

We expect a significant performance improvement by the mixed SC protocol, simply because using HE one multiplication can be implemented using one operation (albeit an expensive one) as in an arithmetic circuit while Yao's protocol works on binary circuits where integer multiplication must be (cumbersomely) emulated using $O(n^2)$ operations (gates). This insight of cleverly combining arithmetic and binary circuits has also been noted in [23]. Table 2 shows the runtimes of both protocols for 16 and 32 bits of input length. Figure 2 shows the same results as a graph. As anticipated the mixed SC protocol clearly outperforms Yao's protocol.

5 Conclusion

We have presented the L1 language intended to implement mixed-protocol SC. It supports different general SC protocols, such as using secret shares [6, 15], homomorphic encryption [11, 12] and garbled circuits [4]. It also supports special SC protocols designed for important problems.

The intention of mixed SC protocols is to improve performance and we tested this hypothesis using two case studies. In one case study a special SC protocol was compared to a general SC protocol and in the other case study a single SC protocol was compared to a mixed SC protocol. In both cases L1 provides superior performance.

We used L1 internally for a number of experiments and implementations of SC. Future work is to continually enhance to it cover different techniques, encryption schemes and protocols. Its built-in benchmarking sub-system allows to perform experiments efficiently leading to novel insights on the practical performance of SC protocols.

5.1 Future Work

We intend to develop on top of L1 another, more high level programming language for cryptographic protocols, called L2. L2 will support more abstract primitives like secure data types, with their operations compiled into a mixed protocol in L1 optimized for performance. We also aim to introduce in L1 support for automatic security verification of the cryptographic protocol programs.

References

1. Kerschbaum, F.: Practical privacy-preserving benchmarking. In: 23rd IFIP International Information Security Conference. (2008)
2. Kerschbaum, F., Dahlmeier, D., Schröpfer, A., Biswas, D.: On the practical importance of communication complexity for secure multi-party computation protocols. In: 24th ACM Symposium on Applied Computing. (2009)
3. Bogetoft, P., Christensen, D.L., Damgard, I., Geisler, M., Jakobsen, T., Kroigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M., Toft, T.: Secure multiparty computation goes live. In: 13th International Conference on Financial Cryptography and Data Security. (2009)
4. Yao, A.: How to generate and exchange secrets. In: In Proceedings of the 27th IEEE Symposium on Foundations of Computer Science. (1986) 162–167
5. Goldreich, O., Micali, S., , Wigderson, A.: How to play any mental game. In: Proceedings of the 19th Symposium on the Theory of Computing. (1987) 218–229
6. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault tolerant distributed computation. In: Proc. of 20th ACM Symposium on Theory of Computing (STOC). (1988) 1–10
7. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols. In: STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing. (1990) 503–513

8. Damgard, I., Jurik, M.: A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In: Proceedings of International Conference on Theory and Practice of Public-Key Cryptography, Lecture Notes in Computer Science 1992. (2001) 119–136
9. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: 41st Annual ACM Symposium on Theory of Computing. (2009)
10. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Proceedings of EUROCRYPT, Lecture Notes in Computer Science 1592. (1999) 223–238
11. Cramer, R., Damgard, I., Nielsen, J.: Multiparty computation from threshold homomorphic encryption. In: Proceedings of EUROCRYPT, Lecture Notes in Computer Science 2045. (2001) 280–299
12. Goethals, B., Laur, S., Lipmaa, H., Mielikäinen, T.: On private scalar product computation for privacy-preserving data mining. In: 7th International Conference on Information Security and Cryptology. (2004)
13. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - a secure two-party computation system. In: Proceedings of the USENIX security symposium. (2004) 287–302
14. Virtual Ideal Functionality Framework: http://www.viff.sk (2010)
15. Cramer, R., Damgard, I., Maurer, U.: General secure multi-party computation from any linear secret sharing scheme. In: Eurocrypt. (2000)
16. Sharemind: http://sharemind.cs.ut.ee/wiki/ (2010)
17. Ben-David, A., Nisan, N., Pinkas, B.: Fairplaymp: a system for secure multi-party computation. In: CCS '08: Proceedings of the 15th ACM conference on Computer and communications security. (2008) 257–266
18. Goldwasser, S.: Multi-party computations: Past and present. In: 16th ACM Symposium on Principles of Distributed Computing. (1997)
19. Nielsen, J.D., Schwartzbach, M.I.: A domain-specific programming language for secure multiparty computation. In: PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security, New York, NY, USA, ACM (2007) 21–30
20. Deitos, R., Kerschbaum, F.: Improving practical performance on secure and private collaborative linear programming. In: DEXA Workshops. (2009) 122–126
21. Aggarwal, G., Mishra, N., Pinkas, B.: Secure computation of the k th-ranked element. In: EUROCRYPT. (2004) 40–55
22. Damgard, I., Thorbek, R.: Efficient conversion of secret-shared values between different fields (2008)
23. Kolesnikov, V., Sadeghi, A.R., Schneider, T.: Modular design of efficient secure function evaluation protocols. Cryptology ePrint Archive, Report 2010/079 (2010)