FairplayMP – A System for Secure Multi-Party Computation

Assaf Ben-David*
School of Engineering and Computer Science
The Hebrew University
Jerusalem 91904, Israel

Noam Nisan*
Google Tel Aviv and
School of Engineering and Computer Science
The Hebrew University
Jerusalem 91904, Israel

Benny Pinkas†
Department of Computer Science
University of Haifa
Haifa 31905, Israel

ABSTRACT

We present FairplayMP (for "Fairplay Multi-Party"), a system for secure multi-party computation. Secure computation is one of the great achievements of modern cryptography, enabling a set of untrusting parties to compute any function of their private inputs while revealing nothing but the result of the function. In a sense, FairplayMP lets the parties run a joint computation that emulates a trusted party which receives the inputs from the parties, computes the function, and privately informs the parties of their outputs. FairplayMP operates by receiving a high-level language description of a function and a configuration file describing the participating parties. The system compiles the function into a description as a Boolean circuit, and perform a distributed evaluation of the circuit while revealing nothing else.

FairplayMP supplements the Fairplay system [16], which supported secure computation between two parties.

The underlying protocol of FairplayMP is the Beaver-Micali-Rogaway (BMR) protocol which runs in a constant number of communication rounds (eight rounds in our implementation). We modified the BMR protocol in a novel way and considerably improved its performance by using the Ben-Or-Goldwasser-Wigderson (BGW) protocol for the purpose of constructing gate tables. We chose to use this protocol since we believe that the number of communication rounds is a major factor on the overall performance of the protocol.

We conducted different experiments which measure the effect of different parameters on the performance of the system and demonstrate its scalability. (We can now tell, for example, that running a second-price auction between four bidders, using five computation players, takes about 8 seconds.)

*Supported by a grant from the Israeli Science Foundation.
†Supported by the Israel Science Foundation (grant number 860/06).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.

Copyright 2008 ACM 978-1-59593-810-7/08/10 ...$5.00.

1. INTRODUCTION

We present FairplayMP, a generic system for secure multi-party computation. This is an extension of the Fairplay system which supported secure computation by two parties [16]. The extension to the multi-party case is needed since cryptographic protocols for the multi-party scenario are completely different than protocols for the two-party case. The system includes two components. The first component is a compiler that enables users to describe a function as a program in a high level language, SFDL 2.0, which is an extension of the SFDL language used in the two-party Fairplay package. The compiler translates this program to a representation as a Boolean circuit. The second component of the system is a cryptographic engine which executes a protocol that computes a circuit securely. The protocol is based on a protocol by Beaver, Micali and Rogoway (the BMR protocol) [2], with enhancements which greatly improve its efficiency.

The goal of secure computation is to enable different parties, each with its own private input, to compute a function of their joint inputs without revealing any information except for the value of the function (and anything implied by it). The canonical example of secure computation is the "millionaires problem", where two millionaires wish to find out which one of them has more money, but must do that without involving any other party and without revealing their actual worth to each other. In more detail, the setting of secure *multi-party* computation consists of n parties (or players) with private inputs $x_1, x_2, \ldots, x_n$ (where x_i is the private input of party i) that wish to jointly compute the functions $f_i(x_1, ..., x_n)$, where player i receives the output $f_i(x_1, ..., x_n)$. If there were a trusted party, then all parties could have sent their inputs to it, and the trusted party could have privately sent each $f_i(x_1, ..., x_n)$ to player i. In this setting it is clear that player i learns nothing but its designated output. Our goal is to have the players alone perform a joint computation in which each player learns its designated output, and no other information is leaked to the players. Namely, they should obtain the same level of privacy they have in a computation that is run by the trusted party.

1.1 Motivation

Elections and auctions are just two of many examples of applications of multi-party computation. Today, most of those computations are done by a trusted party (e.g., an auctioneer). The use of secure multi-party protocols can enable running such applications without any trusted party. (See [17, 4] for a discussion on the advantages of limiting the

knowledge that is learned by auctioneers, and a description of secure protocols for conducting auctions.)

Starting with the seminal work of Yao on secure two-party computation [19], and of [3, 6, 12] on secure multi-party computation, there have been many theoretical constructions of secure computation protocols. There are, however, almost no systems which enable programmers who are not experts in the theory of secure computation to implement such protocols (two notable exceptions are the Fairplay and SIMAP systems, which are described below). Consequently, if one wishes to implement a system for secure computation then he or she need to read the relevant papers and implement the system from scratch. This requirement imposes, of course, a huge barrier for anyone wishing to use secure computation. The goal of our work is to change this situation for the case of multi-party computation.

1.2 Secure Computation and Related Work

We will only discuss here *generic* solutions for secure computation, namely protocols which can be used for computing any function (rather than compute a specific application). For detailed definitions of secure computation see, e.g., the work of Goldreich [11],

Semi-honest adversaries.

We will focus on protocols which provide security against semi-honest adversaries (also known as "honest but curious" adversaries). Adversaries of this type are assumed to follow the instructions that are prescribed for them by the protocol. They may, however, try to learn additional information from the messages that they receive. A stronger type of corruption is performed by adversaries that are denoted as *malicious* adversaries. These adversaries can operate in an arbitrary way and do not have to obey the protocol. It is, of course, easier to ensure security against semi-honest adversaries. The current version of FairplayMP provides security against semi-honest adversaries, future versions will handle the case of malicious adversaries (even though, as we detail below, that case is much more complex).[1]

Security against semi-honest adversaries might be justified if the parties participating in the protocol are somewhat trusted (say, if they are different institutions or agencies that need to compute a function of some information that regulations prevent them from sharing). This level of security is also justified if we trust the participating parties at the time they execute the protocol, but suspect that at a later time an adversary might corrupt them and get hold of the transcript of the information received in the protocol.

We note that protocols secure against malicious adversaries are considerably more costly than their semi-honest counterparts. For example, the generic method of obtaining security against malicious adversaries is through the GMW compiler [12] which adds a zero-knowledge proof for every step of the protocol. Known implementations of secure protocols provide only security against semi-honest adversaries (as is the case with SIMAP), or provide limited security against malicious adversaries (as is the case with the two-party version of Fairplay, which increases the overhead by a factor of c in order to reduce the cheating probability of malicious adversaries to $1/c$). Recent work [15], describes a highly optimized implementation of a two-party protocol secure against malicious adversaries based on a recent efficient construction of Lindell and Pinkas [14], but protocols with similar efficiency do not currently exist for the multi-party case. Even that implementation introduces a considerable performance penalty, as it must substantially increase the number of inputs and the size of the circuit, and must compute multiple copies of the circuit (say, 160 copies in order to limit the error probability to 2^{-40}). Given this difficulty in achieving security against malicious adversaries, the current version of FairplayMP handles only the semi-honest case.

[1]An alternative new security model considers "covert adversaries" [1]. Adversaries of this type may deviate from the protocol definition in order to cheat, but do not wish to be identified as cheaters. This model might be relevant in scenarios where the penalty for being caught cheating is very high. The covert adversaries model is quite attractive since it is usually easier to provide security against covert adversaries than it is to ensure security against malicious adversaries.

Constant round protocols.

We speculate that a major bottleneck of secure computation is the number of communication rounds. Namely the number of times that players need to wait for information sent from other players in order to continue in the computation. (The overhead of starting a communication round is caused by the overhead of the communication infrastructure, and also from the fact that in each round all parties need to wait for the slowest party to conclude its work before they can begin the next round.) For this reason we chose to implement the BMR protocol, which runs in a constant number of rounds, independently of the function that is being computed. (Other protocols need to perform a communication round for every level of the circuit representing the computed function.) Our implementation of FairplayMP uses 8 communication rounds.

Feasibility results in secure computation.

The first generic solution for secure *two-party* computation was presented by Yao [19]. It runs in a constant number of communication rounds and is based on using an enhanced trapdoor permutation. For the *multi-party* case (namely, secure computation involving more than two parties), the protocols of Ben-Or, Goldwasser and Wigderson (BGW) [3] and Chaum, Crépeau, and Damgard (CCD) [6] are based only on the assumption that a private communication channel exists between each pair of parties. They require the computed function to be represented as an arithmetic circuit with gates implementing addition and multiplication over a finite field. The multi-party protocols of Goldreich, Micali and Wigderson [12], and Beaver, Micali and Rogoway (the BMR protocol) [2] are based on representing the function as a Boolean circuit. The BMR protocol runs in a constant number of rounds and therefore we have chosen to implement it. (A newer constant-round protocol for secure multi-party computation was suggested by Damgård and Ishai [8]. It is, however, quite a bit more complex than the BMR protocol.)

Implementations of secure computation.

Other notable implementations of secure protocols are the Fairplay system [16], implementing secure two-party computation, and the SIMAP system [4], which is focused on implementations of auctions using multi-party computation.

1.3 Paper Structure

Section 2 provides an overview of the system and of the high-level language used by it. Section 3 describes the BMR protocol and our enhancements to this protocol. Section 4 describes the protocol that we implemented. Section 5 describes the experiments we conducted to explore the performance of the system.

2. SYSTEM OVERVIEW

The overall structure of the FairplayMP system is similar to that of the Fairplay system [16] and its use the following steps:

1. Users write a program in the high-level programming language SFDL 2.0.
2. This program is compiled into a low level representation as a Boolean circuit.
3. Users write a configuration file describing the ip addresses of the different parties, and different other settings which are needed for the program execution.
4. The program executes the secure multi-party computation of the Boolean circuit in two steps:
   (a) A garbled circuit is created from the Boolean circuit according to the BMR protocol (see below), and a garbled version of the input is generated from the original input.
   (b) The garbled circuit is evaluated by the players who are supposed to receive the output of the program (the system supports providing each of these players with a different output).

2.1 The Secure Function Definition Language

As the starting point of any attempt at security is a clear definition of the requirements, FairplayMP provides a high-level language for specifying the intended input, output and computation. The system uses an augmented version of the SFDL language (Secure Function Definition Language), that was designed for two-party computation and implemented in the Fairplay system [16]. The augmentation – version 2.0 – adds multi-player capabilities as well as several other minor new features. Programs in SFDL define a computation that is carried out by a virtual "trusted party". The SFDL compiler translates the SFDL program into a low-level description in a form of a Boolean circuit (defined in a language called SHDL, which is described in [16]). When writing an SFDL program, one need not care how the trusted third party will be emulated by the participating players, but rather only describe the function computed by the program and the exact intended information input and output.

2.1.1 An example program

Following is an example of a program written in SFDL 2.0, with comments documenting its operation.

```
/**
  Second Price Auction:
  Performs a 2nd price auction between 4
  bidders. Only the winning bidder and the
  seller learn the identity of the winner.
  Everyone knows the 2nd highest price.
**/
program SecondPriceAuction{
  const nBidders = 4;
  type Bid = Int<8>; //enough bits for a bid
  // enough bits to represent a winner.
  type WinningBidder = Int<3>;
  type SellerOutput =
  struct{WinningBidder winner,
         Bid winningPrice};
  //Seller has no input
  type Seller = struct{SellerOutput output};
  type BidderOutput =
  struct{Boolean win, Bid winningPrice};
  type Bidder =
  struct{Bid input, BidderOutput output};
  function void main(Seller seller,
                     Bidder[nBidders] bidder){
    var Bid high;
    var Bid second;
    var WinningBidder winner;
    winner = 0; high = bidder[0].input;
    second = 0;
    // Making the auction.
    for(i=1 to nBidders-1){
      if(bidder[i].input > high){
        winner = i;
        second = high;
        high = bidder[i].input;
      }
      else
        if(bidder[i].input > second)
          second = bidder[i].input;
    }
    // Setting the result.
    seller.output.winner = winner;
    seller.output.winningPrice = second;
    for(i=0 to nBidders-1){
      bidder[i].output.win = (winner == i);
      bidder[i].output.winningPrice = second;
    }
  }
}
```

2.1.2 SFDL Overview

This subsection provides a very short overview of the SFDL language. More details can be found in [16] which describes SFDL 1.0, and a full specification is available on the project's web site at: `http://www.cs.huji.ac.il/project/Fairplay`.

An SFDL program contains two conceptual parts: the first defines data types that are used in the computation, and the second defines the computation itself.

The datatype definitions follow a simple C-like syntax (with a trace of Pascal). It allows booleans, integers whose length is an arbitrary number of bits, enumerated types, and composition using arrays and structures. Pointers are not allowed. Beyond the usual role that datatypes have, they have an important additional role here: the input and output of each participant must be formailized as such a data type. Declared types of this format are used in the header of the main function of the program. Thus, for example, in the auction program above, a "Bidder" has an 8-bit integer input and his output is composed of two parts: a boolean specifying whether this bidder won, and an 8-bit output specifying the winning price. Had we also wanted, for example, to let the bidders learn the identity of the winner, we would have needed to declare the "BidderOutput" structure to also have a "WinningBidder" entry to hold that information.

The second part of the SFDL program is composed of code that specifies the computation itself. The code is organized as a sequence of functions. The main functionality of the program is the evaluation of the last function defined.

This function must be called "main" and receives the players' types and names as its parameters. Thus, for example, the Auction program above, specifies 5 participants: a single "Seller" and 4 "Bidder"s. Each of the participants has its input and output formats defined by its type, and one of them may be empty (e.g. the "Seller" in the example above has no input data).

While the syntax of the code is quite standrad, there are a few significant limitations forced by the fact that this code is compiled into a boolean circuit. In particular, recursion and pointers are not allowed and all loop bounds must be compile-time constants. Also, there is a huge performance gap between an efficient array access when the index is a compile time constant (including for-loop indices – as in the Auction example above), and the non-efficient access oherwise. We refer the reader to the first Fairplay paper [16] for a full discussion as well as the compilation strategy.

2.1.3 Changes from SFDL 1.0

The main difference from the previous version is a change in the format of the "main" function header that now allows an unlimited number of players. This was demonstrated in the Auction program above.

The other changes are quite minor, of which we will only shorty mention allowing access to specific bits in an integer using an array-like notation and the introduction of "generic functions" where the type of the return value of a function depends on the types of the function parameters. For example, we might want to write a function that shifts any integer value, regardless of its size, and thus the return value should be of the same size as the input. This can be done as follows:

```
function generic shiftRight (Int<> v):
                                   Int<v.bitSize> {
  for (i = 0 to v.bitSize-2)
    shiftRight[i] = v[i+1];
  shiftRight[v.bitSize-1] = 0;
}
```

2.2 The Secure Evaluation Cryptographic Engine

Once the intended function was specified in SFDL and compiled into an intermediate low level circuit format, it can now be executed securely by our evaluation engine. The exact configuration and security parameters are specified in a simple XML configuration file (see below), and beyond that the security is ensured.

We implemented the protocol of Beaver, Micali and Rogaway (BMR) [2] with different enhancements which are described in Section 3.2. We chose to implement this protocol for two main reasons:

- We suspected that the number of communication rounds is a major factor of the run time of the protocol. The BMR protocol is one of very few multi-party protocols that run in a constant number of communication rounds, indenpendetly of the function being evaluated or the number of parties.

- The BMR protocol operates on a Boolean circuit representation of the computed function. Other protocols operate on a representation of the function in the form of an arithmetic circuit, whose atomic gates implement addition and multiplication in a finite field. We preferred the Boolean circuit representation since it supports efficient comparisons which we believe is an important key in all "real life" programs.

We separated the computation process to three different processes, each representing a role of a player in the computation: those players contributing an input, those participating in the computation and those learning an output (this is in contrast to the original definitions of multi-party computation where each player participates in all roles). Each player only participates in computations and interactions relevant to its roles (but a player can participate in several roles without compromising security). The number of players that build the "garbled circuit" and emulate the "trusted third party" is no longer dependent on the number of input or output players. This allows players to "distribute" trust among chosen players who are not necessarily related directly to the function (as input/result players).

To recap, the three supported player types are:

- Input players (IP) - The players which contribute input for the computation.

- Computation players (CP) - The players which build the garbled circuit and emulate the trusted third party. (The protocol will be secure as long as an adversary cannot corrupt half or more of the CPs. Therefore it is reasonable to envision scenarios with a large number of IPs but only, say, about ten CPs, implemented by more trusted parties.)

- Result players (RP) - The players which receive some output out of the circuit.

We conducted different experiments which are described in Section 5 and which demonstrate the scalability of the system. In particular, they show that the depth of the circuit does not affect the overhead.

The FairplayMP package is available for download on the project's web site.

2.3 The configuration file

The system uses a configuration file (config.xml) which has two main purposes. The first is to coordinate between all the computers involved in the computation (i.e. the computers that provide input, perform the computation and provide output – each of them must have a copy of this file). In this part of the xml file, the name of the circuit file and a list of the participating IP addresses are given. The second purpose is to set all the security parameters needed for the computation, such as the pseudo-random generator (PRG) to use, port numbers, the certificate store that will be used for SSL connections and the prime number that will be used as a modulus in the computation (the length of this modulus is related to the security parameter k – it must be at least k times the number of CPs plus 2). The full version of this paper includes an example configuration file.

2.4 The interface between the two system parts

The FairplayMP package is composed of two components: the SFDL 2.0 compiler and the FairplayMP cryptographic engine executing the secure multi-party computation. Although the package is intended to allow users to easily use the combined functionality, write a function and compute it, some users might want to use only one part of the system. For this reason we created a standard interface between

the two parts of the system, in fact both a data interface and a programmatic api. (Our own implementation indeed goes through these two interfaces.) The data interface is in SHDL (Secure Hardware Definition Language) that contains a specification of the function and its inputs and outputs as a low-level boolean circuit. The programmatic interface is a Java-language Intreface (defined in Circuit.java) which directly interfaces to the cryptographic engine. The full documentation and source code are available on the project's web site at: `http://www.cs.huji.ac.il/project/Fairplay`.

These interfaces allow either using our cryptographic engine to evaluate functions that were specified by means other than SFDL (e.g. optimized by hand), or to use the SFDL language with a different (boolean-circuit-based) cryptographic engine.

3. THE BMR PROTOCOL

We first describe the basic properties of the protocol of Beaver, Micali and Rogaway (the BMR protocol) [2], and then describe our enhancements to this protocol. The details of the BMR protocol appear in the full version of our paper.

3.1 The Basic BMR Protocol

The BMR protocol [2] implements secure multi-party computation in a constant number of rounds. It is essentially a multi-party version of Yao's two-party protocol. The protocol has two phases. During the first phase a "garbled circuit" and a matching set of "garbled inputs" are created by the input players and the computation players. Then, in the second phase, each of the result players evaluates the garbled circuit individually. We are only concerned in this paper with security against semi-honest adversaries, and therefore are able to present a protocol which is simpler than the original protocol (which uses generic zero-knowledge proofs to provide security against malicious adversaries). The protocol is based on the following two principles:

Hiding values of internal wires: Two random seeds are attached to each wire in the circuit (in Yao's paper [19] these seeds are referred to as garbled values, but we also refer to them as seeds since they are used as an input to a pseudo-random generator). Each seed is actually defined as the concatenation of seeds generated by each of the parties (namely, seed s_i is defined as $s_i = s_i^1 \circ \cdots s_i^n$). One of the seeds of a wire represents the value 0 and the other seed represents the value 1. For each gate the seeds of the output wire are hidden using the seeds of the input wires, according to the truth table of the gate. Consequently, knowledge of one seed for each of the two input wires of a gate enables to uncover a single seed of the output wire, corresponding to the appropriate output derived from these input values.

Randomizing the 0/1 values: Using random seeds instead of the original $0/1$ values does not hide the original value if it is known that the first seed corresponds to 0 and the second seed to 1. Therefore, an unknown random bit, denoted by λ, is assigned to each wire. We define the *external value* of each wire as the exclusive-or of the *real value* (or *original value*) of the wire and the λ value attached to the wire. This enables the evaluation process to use the external value of the wire while concealing its real value. A party participating in the protocol might know the seed matching the external value 1 (and know that the external value is 1), but without knowing λ it gains no information about the real value of the wire.

The first phase of the protocol (where the parties construct the garbled circuit) is computed using a secure multi-party computation that is done separately, but in parallel, for each gate. (This is the major bottleneck of the computation. We were able to improve its run time, as is detailed in Section 3.2.) In the second phase, the result players receive information which enables them to compute the output of the function. The entire protocol runs in a constant number of communication rounds.

3.2 Our Enhancements to the Basic BMR Protocol

Constructing the tables.

In order to simplify the secure computation of the truth tables, we replaced in this computation the exclusive-or operation (which is used is order to hide the seed $s_i^1 \circ \cdots \circ s_i^n$ of an output wire by "exclusive-or"-ing it with the expansion of the seeds of the input wires) with an addition operation in a finite field. This enhancement allows us to use the BGW protocol (due to Ben-Or, Goldwasser and Wigderson [3]) which is more efficient as it can apply atomic operations to the entire *string* $s_i^1 \circ \cdots \circ s_i^n$ which is a single item in the field. Of course, this requires that items in the finite field be longer than $|s_i^1 \circ \cdots \circ s_i^n|$. This requirement should typically not be a problem, since for n CPs and for seeds of length m the modulus should only be of length $nm + 1$. The number of servers n is pretty low (say, 10 or less servers), and m is typically in the range 64-128, and therefore the length of the modulus should not exceed lengths which are common in cryptography.

The shares of the string $s_i^1 \circ \cdots \circ s_i^n$ itself can be generated in a distributed way: Party j chooses s_i^j and shifts it to the right $k \cdot (n - j)$ times. It sends shares of this string to the parties. The sum of the shares that each party receives is a share of $s_i^1 \circ \cdots \circ s_i^n$.

The computation of the table is then completed in two computation steps. In order to compute the correct value of the table (of a gate with input wires a, b and output wire c) the protocol first computes shares of $\lambda_a\lambda_b$ and then computes $\lambda_a \bigotimes \lambda_b$ from the three shared values: λ_a, λ_b and $\lambda_a\lambda_b$. Then the protocol computes the value $(\lambda_c - \lambda_a\lambda_b)^2$, which is either 1 or 0, depending on whether $\lambda_c = \lambda_a\lambda_b$. This value is multiplied by the concatenation of the appropriate seed values to obtain the right value for the table entry.

Collective coin flipping.

Each λ value must be a random bit which is shared among the computation players using addition in a finite field. Generating such a bit is easy in a field of characteristic 2. We need, however, to generate this bit is a finite field with a larger characteristic. This is done using a recent protocol due to [7], which is based on arithmetic operations that are implemented using the BGW protocol. The protocol is secure against coalitions of up to $\lfloor n/2 \rfloor$ corrupt players. It is described in the full version of our paper, and also in the details of the implemented protocol in Section 4.

Player types.

The original BMR paper assumed that all players have inputs, all of them participate in the computation, and all of them should learn an output. We define three possible types of players: input players (IPs), computation players (CPs) and output/result players (RPs). A player can possibly have

several of these roles. A typical scenario might involve many input players and only a handful of computation players. Security is ensured as long as the adversary cannot corrupt more than a strict minority of the computation players. (It can corrupt any number of input or result players). The computation players should therefore be more trusted, and be chosen in a way which supports the assumption that it is unlikely that half or more of them are corrupt.

With regards to result players, we make sure that each result player receives from the computation players only the bits that are relevant to its output. This ensures in a simple way that each result player can only recover its own output and learns nothing else (in other protocols, on the other hand, encryption might be needed to ensure this property).

Handling large fan-out.

The output of a gate might be input to *several* other gates (namely, the gate might have fan-out greater than 1). The protocol must ensure, however, that the single seed value is expanded to different values in every gate. The expansion of the seed is therefore done using a cryptographic function F which has two inputs: the seed of the input wire, and the index of the gate to which the wire enters. As a result, different g and h values are generated for every gate in which the wire is used.[2]

4. THE IMPLEMENTED PROTOCOL

We decided to implement the program in Java in order to support easy cross-platform usage. The focus in our implementation was on performance in terms of the number of communication rounds and the size of the messages (note that no public key operations are needed, and therefore the computation costs does not necessarily dominate the communication overhead). FairplayMP separates the code to four main packages that work together in a modular way:

The *player package* holds the three different types of players which implements the Player interface. Each of these runs as a different thread in order to prevent a bottleneck from occurring when one computer is participating in more then one role.

The *communication package* holds the server and client threads which use SSL encryption in a peer to peer communication model. It also contain the Msg and MsgCP class which define a message in the protocol.

The *circuit package* holds the Circuit interface and the SHDL Circuit implementation.

The *utils package* includes the rest of the classes, including the implementation of the pseudo-random generator and of the BGW [3] protocol.

4.1 Players

We denote the number of computation players by n. Below we list the tasks implemented by each type of players.

[2]In the theoretical analysis of the system we cannot model the cryptographic function F as a simple PRG, since a PRG provides a pseudo-random output only if its input is random (and the gate id is not random). The function F can be modeled as a pseudo-random function (PRF) whose key is the seed and whose input is the gate index (it is well known how to construct a PRF from a PRG). In FairplayMP we implement F by applying a hash function such as SHA-1 to the exclusive-or of the seed and the gate index.

4.1.1 Input players (IP)

Each IP performs the following tasks for each wire w of its *input* wires, which has an input value b_w:

1. Creates a random bit λ_w and random seeds of length nk: s^i_{2w} and s^i_{2w+1} (representing 0 and 1, respectively).
2. Shares λ_w using $(\lfloor n/2 \rfloor + 1)$-out-of-$n$ secret sharing between all the computation players. The seeds s_{2w} and s_{2w+1} are also split to n concatenated seeds of length k (s_{2w} is split into $s^1_{2w} \circ \cdots \circ s^n_{2w}$ and s_{2w+1} is split in a similar way). The k-bit seeds s^i_{2w} and s^i_{2w+1} are sent to the ith computation player.
3. Sends the seed $s_{(b_w \oplus \lambda_w)}$ and the bit $(b_w \oplus \lambda_w)$ to the result players. (These values correspond to the external value of this wire, and since the value of λ_w is secret, the real value is kept hidden.)

4.1.2 Computation players (CP)

1. For each wire w (which is not an input wire) each computation player i generates two random seeds of length k: s^i_{2w} and s^i_{2w+1} (representing 0 and 1, respectively). It then shifts each of these seeds to the right $k \cdot (n-i)$ times. It sends to the other computation players shares of the resulting strings, using $(\lfloor n/2 \rfloor + 1)$-out-of-$n$ secret sharing. (Note that for every index v the sum of the shares that each party receives is a share of $s_v = s^1_v \circ \cdots \circ s^m_v$, which is string of length nk.)

   Let us denote by ev_v the external value of wire v, namely the result of the exclusive-or of λ_w and of the real value of the wire (the Binary value of the wire when the circuit is fed with the input of the players).

   The result players need to know the external values of all wires. We therefore ask the computation players to compute and use the value $s_v = s^1_v \circ \cdots \circ s^m_v \circ ev_v$. This is done in the following way: each computation player sums the shares of s_{2w}, and similarly it sums the shares of s_{2w+1} that it received. It then multiplies each of these results by 2. Finally, it adds 1 to the latter of these two results.

   For every wire w, the computation players also engage in a secure computation in which each of them computes a share of a random bit λ_w (according to the method listed in Section 3.2). (Overall, this step requires two messages.)
2. If the input wires of a gate are wires α and β and the output wire is wire γ we denote $a = 2\alpha$, $b = 2\beta$ and $c = 2\gamma$ (therefore for wire α the value 0 is represented by s_a and the value 1 by s_{a+1}, and the same holds for β and γ with respect to b and c.)

   The parties then collectively compute the entries of the truth tables of a gate which implements the Binary operation $\bigotimes$, according to the following definition:

$$A_g = g^1_a + \cdots + g^n_a + g^1_b + \cdots + g^n_b + \begin{cases} s^1_c \circ \cdots \circ s^n_c \circ 0 & \text{if } \lambda_\alpha \bigotimes \lambda_\beta = \lambda_\gamma \\ s^1_{c+1} \circ \cdots \circ s^n_{c+1} \circ 1 & \text{otherwise} \end{cases}$$

$$B_g = h^1_a + \cdots + h^n_a + g^1_{b+1} + \cdots + g^n_{b+1} + \begin{cases} s^1_c \circ \cdots \circ s^n_c \circ 0 & \text{if } \lambda_\alpha \bigotimes \overline{\lambda_\beta} = \lambda_\gamma \\ s^1_{c+1} \circ \cdots \circ s^n_{c+1} \circ 1 & \text{otherwise} \end{cases}$$

$$C_g = g_{a+1}^1 + \cdots + g_{a+1}^n + h_b^1 + \cdots + h_b^n + \begin{cases} s_c^1 \circ \cdots \circ s_c^n \circ 0 & \text{if } \overline{\lambda_\alpha} \bigotimes \lambda_\beta = \lambda_\gamma \\ s_{c+1}^1 \circ \cdots \circ s_{c+1}^n \circ 1 & \text{otherwise} \end{cases}$$

$$D_g = h_{a+1}^1 + \cdots + h_{a+1}^n + h_{b+1}^1 + \cdots + h_{b+1}^n + \begin{cases} s_c^1 \circ \cdots \circ s_c^n \circ 0 & \text{if } \overline{\lambda_\alpha} \bigotimes \overline{\lambda_\beta} = \lambda_\gamma \\ s_{c+1}^1 \circ \cdots \circ s_{c+1}^n \circ 1 & \text{otherwise} \end{cases}$$

In order to perform this computation, each CP i operates as follows:

(a) For each gate:

i. For each seed of each input wire it creates the values g, h by applying the cryptographic function F to the seed and to the index of the gate which it enters (see Footnote 2 above for a description of the modeling and implementation of F). The player computes for example, for the first seed of wire α, the values $g_a^i \mid h_a^i = F(s_a^i, \text{gate id})$. The length of the output of F is $|g_a^i| + |h_a^i| = 2|\mathcal{F}|$, where $\mathcal{F}$ is the finite field which is used by the system.

ii. It sends shares of the results of the previous step to all other CPs. Secret sharing is done using a threshold of $(\lfloor n/2 \rfloor + 1)$-out-of-$n$.

iii. It computes the truth table of the gate. For example, for entry A_g it performs the following operations (similar operations are applied to the other entries):

A. Adds the shares of $g_a^1, \ldots, g_a^n$ and of $g_b^1, \cdots, g_b^n$ that it received.

B. Applies the multiplication step of the BGW protocol to the shares of λ_α and λ_β, to obtain a share of $\lambda_\alpha \lambda_\beta$ (the share is in $(\lfloor n/2 \rfloor + 1)$-out-of-$n$ secret sharing, as are the previous shares).

C. Computes a share of $\lambda_\alpha \bigotimes \lambda_\beta$ by applying linear operations to the shares that it has of λ_α, λ_β and $\lambda_\alpha \lambda_\beta$ (any Boolean function $\bigotimes$ of $\lambda_\alpha, \lambda_\beta$ can be expressed as a linear combination of λ_α, λ_β and $\lambda_\alpha \lambda_\beta$.)

D. Subtracts the share of λ_γ from the result. This results in a share of 0 if $\lambda_\alpha \bigotimes \lambda_\beta = \lambda_\gamma$ and a share of either 1 or -1 otherwise.

E. Raises the result to the power of two. This provides a share of 0 if there is equality and a share of 1 otherwise. Denote this share as v_i. (Raising to the power of two is implemented as a multiplication step of the BGW protocol.)

F. Multiplies v_i with the share of the seed s_{c+1}, and multiplies $(1-v_i)$ with the share of the seed s_c. The multiplications are done using the BGW protocol.

G. Sums the results of the last step, and adds this value to the shares of g and h that were received. The result is a share of A_g using $(\lfloor n/2 \rfloor + 1)$-out-of-$n$ secret sharing.

3. For all gates, it sends to all the result players the share of $A_g, \ldots, D_g$ from above.

4. It also sends to each result players the shares it has of the λ values associated with the output wires of this result player.

4.1.3 Result players (RP)

1. For each input wire the player receives the string $s_{b_w \oplus \lambda_w}$ and the external value $b_w \oplus \lambda_w$ from the IP which is responsible for the wire.

2. For every gate, it receives shares from every CP and uses them to reconstruct the entries $A_g, \ldots, D_g$ of the truth table of that gate.

3. Initially for each input wire of the circuit, the player holds the seed s_{2w} or s_{2w+1}, and the corresponding external value ($2w$ or $2w + 1$).

   Suppose inductively that the player holds s_{a+p} for the left input wire of a gate, and s_{a+q} for the right input wire, where a and b are even and $p, q \in \{0, 1\}$ are the external values of these wires. Suppose that the output wire of the gate is γ. Let F_L denote the left half of the output of F, and let F_R denote its right half. Then it should expand each part of the seed (for each computation player) by computing the following functions: $\forall 1 \leq i \leq n$ $g_{a+p}^i = F_L(s_{a+p}^i, g)$, $h_{a+p}^i = F_R(s_{a+p}^i, g)$, $g_{b+q}^i = F_L(s_{b+q}^i, g)$, $h_{b+q}^i = F_R(s_{b+q}^i, g)$. It can then compute the right seeds for γ by computing:

$$\sigma_\gamma = \begin{cases} A_g - g_{a+p}^1 - \ldots - g_{a+p}^n - g_{b+q}^1 - \ldots - g_{b+q}^n & \text{if } p = 0 \text{ and q=0} \\ B_g - h_{a+p}^1 - \ldots - h_{a+p}^n - g_{b+q}^1 - \ldots - g_{b+q}^n & \text{if } p = 0 \text{ and q=1} \\ C_g - g_{a+p}^1 - \ldots - g_{a+p}^n - h_{b+q}^1 - \ldots - h_{b+q}^n & \text{if } p = 1 \text{ and q=0} \\ D_g - h_{a+p}^1 - \ldots - h_{a+p}^n - h_{b+q}^1 - \ldots - h_{b+q}^n & \text{if } p = 1 \text{ and q=1} \end{cases}$$

   The least significant bit of the string σ_γ that is computed is the external value of the wire γ. The rest of σ_γ contains the concatenation of seeds of this wire.

4. For each output bit w:

   (a) The player reconstructs λ_w according to the shares received from each CP.

   (b) It computes the exclusive-or of λ_w and the external bit of this wire (computed by the gate from which it is output) to get the desired output b_w.

4.2 Security

The protocol is secure against a coalition of at most $\lfloor n/2 \rfloor$ corrupt computation players, as long as they operate in a semi-honest way. Adding any number of corrupt, but semi-honest, input players or result players does not affect the security of the protocol. We will only sketch here the arguments showing the security of the protocol (that are, of course, based on the security of the BMR and BGW protocols). In Step 1 each computation player receives shares of a $(\lfloor n/2 \rfloor + 1)$-out-of-$n$ secret sharing scheme, and engages in the bit-flipping protocol of [7]. It Step 2 it runs the BGW protocol. Therefore, as long as we assume that the BGW protocol and the bit-flipping protocol are implemented securely, a coalition of at most $\lfloor n/2 \rfloor$ computation players learns nothing (in particular, it does not learn the λ values of wires).

5. PERFORMANCE

We performed different experiments to examine the performance of the system. In addition to checking the overall runtime, we examined the effects of different properties of the computed function, such as the number of gates or the depth of the circuit, and the effects of different properties of the computation setting, such as the number of players and mixing strong and weak machines in the same setting. We used both "real life" SFDL 2.0 programs that were translated by the SFDL compiler to circuits, and hard coded java implemented circuits (using the Circuit.java interface).

The setting: All experiments were run in the Hebrew University's CS lab on a grid of computers, each with two Intel Xeon 3GHz CPU processors and 4GB of RAM. (The last experiment used also a different machine, as is detailed below.) The pseudo-random function F was implemented as SHA-1. All results were computed as an average of three different runs on all the computers and measured in milliseconds.

5.1 Experiments

Measuring the general run time.

We checked the run time of computing circuits which are structured as full binary trees. The depth of the circuits ranged from 1 to 10, where the number of gates in a circuit of depth d is 2^d. These circuits were computed in settings with 5,7,9 and 11 computation players (CPs).[3] The security parameter was set to $k = 80$, and the length of the modulus was set to be just long enough for each setting, namely it was equal to k times the number of CPs plus 2 (e.g., 402 bits for a setting with 5 CPs). The results of the experiments appear in Table 1. (The results do not include the run time of input players, which is negligible. The run time of the CPs includes the time spent waiting for messages and receiving them. The run time of the RPs includes only the computation time. We do not detail in this table the run time for circuits of depth 1 to 4, for lack of space.)

We analyze the run time in detail below. Overall, the run time seems to be reasonable. For example, with 5 CPs (namely, providing security against coalition of any two corrupt CPs), computing a circuit with 1024 gates takes about 10 seconds. We show below that the run time grows linearly with the number of gates, and therefore we expect, for example, that a very large circuit with 100,000 gates will be computed in about 1000 seconds, or 16 minutes. This computation is not instantaneous, but its running time seems reasonable for important applications (for example, the privacy-preserving sugar beet auction reported in [4] ran for half an hour, in a setting with only three servers).

Dependency on the size of the circuit.

Theoretically, if we fix the number of players and increase the size of the circuit, we expect the run time of both the computation players (CP) or the result players (RP) to be a linear function of the size of the circuit. Figure 1 plots (based on the results of Table 1) the run time of the CPs and RPs as a function of the circuit size in a setting with 5 CPs.

Conclusion: Our experimental results show that the run time has linear dependency on the size of the circuit.

[3]Recall that a setting with 5 CPs is the minimal setting providing security against collusions of two servers. Security against corrupt servers that do not collude can be achieved with only two servers, using the two-party protocol of Yao [19]. Therefore a setting with 5 CPs is the minimal setting in which it makes sense to use secure multi-party computation.

Depth	5	6	7	8	9	10
Gates (1)	32	64	128	256	512	1024
5 CPs						
CP (2)	2234	2622	3218	4299	6215	9635
RP (3)	64	130	234	440	770	1394
7 CPs						
CP (2)	2684	3358	4480	6281	9682	17095
RP (3)	86	154	295	608	1105	2093
9 CPs						
CP (2)	3278	4380	5915	8835	15293	28045
RP (3)	115	219	438	845	1621	3150
11 CPs						
CP (2)	3921	5575	8000	12925	23678	54497
RP (3)	144	289	569	1123	2198	4276

Table 1: Run time (in msec) of evaluating full binary circuits of different sizes, with security parameter $k = 80$, for different numbers of CPs. The length of the modulus is k times the number of CPs, plus 2. (1) Number of gates in the circuit. (2,3) Average running time of the Computation Player, and the Result Player, respectively.

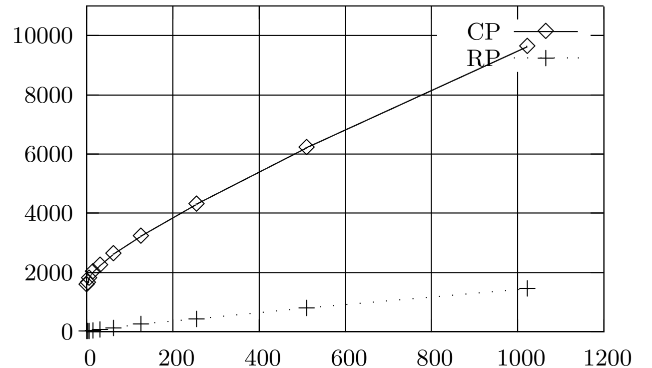

Figure 1: A plot of the running time (in msecs) of the CPs and RPs as a function of the number of gates, in a setting with 5 CPs. The data was taken from Table 1.

Dependency on the number of computation players.

When comparing the run time of computing the same circuit in settings with different numbers of computation players we should recall that part of the computation has linear dependency on the number of players (e.g. computing the gates truth table), while part of the computation has quadratic dependency on the number of players (e.g. executing the interpolation step of the BGW protocol for a polynomial whose degree in linear in the number of players). The communication sent and received by each player is always linear in the number of players. We plot in Figure 2 the run time of computing three different circuits (with 256,

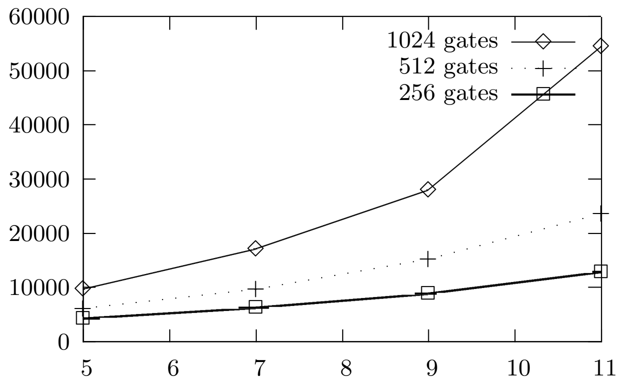

Figure 2: A plot of the run time (in msecs) of the CPs as a function of the number of CPs, for circuits with 256, 512 and 1024 gates, respectively. The data was taken from Table 1.

Depth	32	64	128	256	512	1024
Gates	32	64	128	256	512	1024
CP	2259	2641	3228	4342	5998	9639
RP	65	127	239	435	798	1396

Table 2: The run time of evaluating of a circuit whose depth is equal to the number of gates, in a setting with 5 CPs. (Results should be compared to the first two lines of Table 1.

512 and 1024 gates, respectively, taken from Table 1) as a function of the number of CPs.

Conclusion: The dependency on the number of CPs is greater than linear and is close to quadratic (the quadratic regression correlation coefficient is 0.997 or better). More data points are needed for a more accurate analysis.

Measuring the effect of the circuit depth.

A major advantage of the BMR protocol is that it runs in a constant number of communication rounds and therefore its run time should be independent of the circuit depth. We verified this property by running experiments which evaluated circuits constructed as a linked list (namely, where $\forall 1 \leq i \leq d$ there is exactly one gate in distance i from the root). In experiment i we examined a circuit with 2^i gates and depth 2^i (compared to 2^i gates and a depth of i in experiment i in the previous setting). We ran this experiment with 5 CPs and security parameter $k = 80$.

The results are described in Table 2. A comparison to Table 1 (the part describing the results for 5 CPs) shows that the measured times in the two tables are within 3.5% of each other.

Conclusions: The results demonstrate that, as expected from the BMR protocol, the depth of the circuit has marginal effect on the run time of the protocol. We believe that other protocols for secure multi-party computation would have performed pretty badly in this experiment.

Dependency on the security parameter k.

We measured the effect of changing the security parameter k. (Typically, k should be sufficiently long to be a key for a symmetric encryption function, namely be in the range

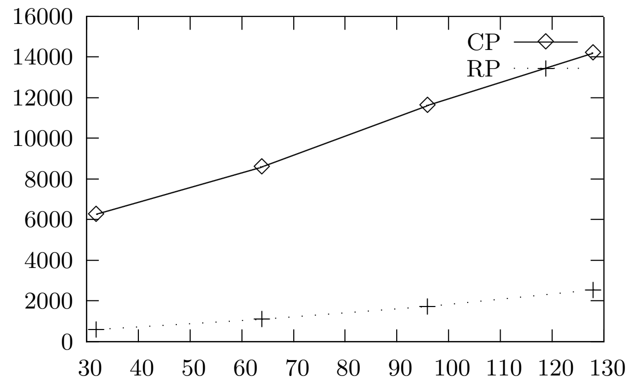

Figure 3: The run time (in msecs) as a function of the security parameter k, in a setting with 5 CPs and a circuit of 1024 gates.

[64, 160]). We ran experiments of computing a circuit with 1024 gates in a setting with 5 computation players, with the security parameter k taking the values $32, 64, 96$ and 128, and the modulus being of length $5k+2$ bits. The results are shown in Figure 3.

Conclusion: The dependency of the run time on k is very close to linear.

"Real life" functions.

We wanted to check whether the system can be used for solving real life problems. For this purpose we experimented with protocols for computing auctions and for voting.

Second Price Auction: We ran this experiment using the example program SecondPriceAuction given in Section 2.1.1. It involves four input players (bidders) with 8 bit bids and one result player (seller). We ran this example with five computation players which were the four bidders and the seller. (Therefore this experiment is similar to the first set of experiments in Table 1 which uses five CPs.) The SFDL program was compiled to a circuit with 400 gates. The measurements of the run time were 5387 msec for the CPs and 640 msec for the RP. These results are comparable to the 512 gate experiment in Table 1. The total run time of the experiment (including all communication delays) was around 8 seconds. We repeated this experiment with 10 bidders, resulting in a compiled circuit with 1380 gates, where five of those bidders were also computation players. The run time was 27306 msec for the CPs and 5908 msec for the RP.

Voting: We ran a simple SFDL program for voting (this program appears in the full version of the paper). The program involves 30 voters and 5 computation players which were run on different machines than those used by the voters . The circuit contained 1383 gates. The run time for the computation players was around 15 seconds. (We suspect that the improvement in the run time compared to that of the second price auction with 10 bidders, which uses a circuit of a comparable size, was the result of the fact that the CPs were run on different computers than those running the IP and RP threads. Some other tests that we made support this explanation.)

Conclusions: Both results are very reasonable and are similar to those of computing complete binary tree with the same number of gates. The running time is proportional to

Depth	5	6	7	8	9	10
Gates	32	64	128	256	512	1024
Strong machine						
CP	5099	6510	9224	14251	23627	41572
RP	62	128	221	426	794	1402
Weak machine						
CP	4876	6414	9444	14679	24653	43310
RP	237	442	903	1614	3122	6233

Table 3: The performance of the weak machine player and the strong machine player in a mixed environment, when evaluating a full binary circuit using 5 CPs, four of which running on strong machines and the other one using a weak machine.

the number of gates in the compiled version of the examples.

Measuring the influence of a less powerful computer.

This experiment examines the increase in run time which is caused by changing the homogeneous setting (where all players are run on strong machines), to a mixed setting where one of the players uses a weaker computer (a Pentium III machine with 512MB of RAM). The experiment repeated the first experiment (Table 1), namely, it used 5 CPs to evaluate a circuit which is a complete binary tree, with security parameter $k = 80$. We detail in Table 3 the performance of the weak and strong machines in this experiment.

Conclusions: In the mixed setting the run time of the RP on the slow machine is of course considerably slower than that of the faster player. This is not surprising since the computational capabilities of the machines are very different. It is more interesting to examine the run time of the computation player (CP). This time is about the *same* on both types of machines (give or take 5%). However, the run time of the fast machine is much slower in this experiment than its run time in a comparable experiment in the homogeneous setting (as is described in the first row of Table 1). This means that the performance of fast machines is dramatically reduced if even a single player is using a weak machine. The reason for this is that in every communication round the fast players have to wait until the weakest player finishes its computation and sends its results.

6. REFERENCES

[1] Y. Aumann and Y. Lindell, Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries In 4*th TCC*, pages 137–156, 2007.

[2] D. Beaver, S. Micali and P. Rogaway. The round complexity of secure protocols. In 22*th STOC,* pp. 503–513, 1990.

[3] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In 20*th STOC,* pages 1–10, 1988.

[4] P. Bogetoft, D.L. Christensen, I. Dåmgard, M. Geisler, T. Jakobsen, M. Krøigaard, J.D. Nielsen, J.B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach and T. Toft. Multi-Party Computation Goes Live Cryptology ePrint Archive, Report 2008/068, 2008.

[5] P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. A practical implementation of secure auctions based on multi-party integer computation. Proc. of Financial Cryptography, LNCS vol. 4107, Springer-Verlag, 2006.

[6] D. Chaum, C. Crépeau and I. Damgård. Multi-party Unconditionally Secure Protocols. In 20*th STOC*, pages 11–19, 1988.

[7] R. Cramer, I. Damgrd and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In 2*nd TCC*, pages 342–362, 2005.

[8] I. Damgård and Y. Ishai. Constant-Round Multi-Party Computation Using a Black-Box Pseudorandom Generator. In *Crypto '2005*, pp. 378-394, 2005.

[9] S. Even, O. Goldreich and A. Lempel. *A Randomized Protocol for Signing Contracts*, Communications of the ACM, vol. **28**, 1985, pp. 637–647.

[10] R. Gennaro, M. O. Rabin and T. Rabin. Simplified VSS and Fast-track Multi-Party Computations with Applications to Threshold Cryptography. In 17*th PODC*, pages 101–111, 1998.

[11] O. Goldreich. *Foundations of Cryptography: Vol. 2 – Basic Applications.* Cambridge University Press, 2004.

[12] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In 19*th STOC,* pages 218–229, 1987.

[13] Y. Lindell and B. Pinkas. A Proof of Yao's Protocol for Secure Two-Party Computation. To appear in the *Journal of Cryptology.* Also appeared as *Cryptology ePrint Archive,* Report 2004/175, 2004.

[14] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT 2007*, Springer-Verlag LNCS 4515, 52–78, 2007.

[15] Y. Lindell, B. Pinkas and N. Smart. Implementing Two-Party Computation Efficiently with Security Against Malicious Adversaries. *6th Conf. on Security and Cryptography for Networks (SCN)*, Springer-Verlag LNCS 5229, pp. 2–20, 2008.

[16] D. Malkhi, N. Nisan, B. Pinkas and Y. Sella. Fairplay – A Secure Two-Party Computation System. 13*th USENIX Security Symposium*, pages 287–302, 2004.

[17] M. Naor, B. Pinkas and R. Sumner. Privacy Preserving Auctions and Mechanism Design. Proceedings of the *1st ACM conf. on Electronic Commerce*, November 1999.

[18] J.D. Nielsen and M.I. Schwartzbach. A domain-specific programming language for secure multi-party computation. Proceedings of Programming Languages and Security (PLAS), 2007, ACM press.

[19] A. Yao. How to Generate and Exchange Secrets. In 27*th FOCS*, pages 162–167, 1986.