Fast and Maliciously Secure Two-Party Computation Using the GPU

Tore Kasper Frederiksen and Jesper Buus Nielsen*

Department of Computer Science, Aarhus University
{jot2re,jbn}@cs.au.dk

Abstract. We describe, and implement, a maliciously secure protocol for two-party computation in a parallel computational model. Our protocol is based on Yao's garbled circuit and an efficient OT extension. The implementation is done using CUDA and yields fast results for maliciously secure two-party computation in a financially feasible and practical setting by using a consumer grade CPU and GPU. Our protocol further uses some novel constructions in order to combine garbled circuits and an OT extension in a parallel and maliciously secure setting.

1 Introduction

Secure two-party computation (2PC) is the area of cryptography concerned with two mutually distrusting parties who wish to securely compute an arbitrary function on their joint and private input without leaking any information. The area was introduced in 1982 by Yao [25], specifically for the *semi honest* case where both parties are assumed to follow the prescribed protocol. Yao showed how to construct such a protocol using a technique referred to as the *garbled circuit approach*. Later, a solution in the *malicious* setting, where one of the parties might deviate from the prescribed protocol in an arbitrary manner, was given in [4]. Another approach for malicious security, called the *cut-and-choose approach*, involves running several instances of garbled circuits in parallel, with some random instances being completely revealed to verify that the other party has behaved honestly. Efficient 2PC and secure multi-party computation (MPC) have many practical applications. The first case of this is described in [2], where MPC was used for deciding the price of a national sugar beet auction in Denmark. Other applications for 2PC and MPC include voting, anonymous identification, privacy preserving database queries etc.

Recently a lot of research has gone into making 2PC efficient enough to be practical, cf. [7,13,15,18,19,21]. Most previous approaches have focused on doing this in a sequential model [13,15,18]. However, the recent evolution of processors seems to indicate a convergence of speed, whereas the amount of cores in processors seem to increase. Thus, constructing algorithms and cryptographic protocols

* Partially supported by the Danish Council for Independent Research via DFF Starting Grant 10-081612. Partially supported by the European Research Commission Starting Grant 279447.

M. Jacobson et al. (Eds.): ACNS 2013, LNCS 7954, pp. 339–356, 2013.

that work well in a parallel model will be paramount for hardware based efficiency increases in the future, which is why we take the parallel approach to increase the speed of 2PC.

Previous work in "parallel cryptography" started with [22], where a cluster of either CPUs or GPUs was used to execute 3072 semi honest protocols for 1-out-of-2 oblivious transfer (OT) followed by gate garbling/degarbling in parallel.[1] In [12] 512 cores of a cluster was used to do OT along with circuit garbling in parallel to achieve malicious security using the cut-and-choose approach. In this manner they managed to use the inherit parallelism of the cut-and-choose approach to achieve very fast and maliciously secure 2PC. Any other work taking a parallel approach to cryptography that we know of focuses either on attacks e.g. [24] or simultaneous applications of more primitive cryptographic computations e.g. [20].

Contributions. Our main contribution is a careful implementation, along with a general protocol, for maliciously secure 2PC using a *Same Instruction, Multiple Data (SIMD)*, or *Parallel Random Access Model (PRAM)* computation device. Our protocol is UC secure in the *Random Oracle Model (ROM)*, *OT-hybrid model* and based on Yao's garbled circuit approach [25] along with the *OT extension* (See Section 2) of [18] and a few novel ideas. Computationally our protocol relies solely on symmetric primitives, except for a few seed OTs used in the OT extension which only need to be done once for each pair of parties. Furthermore, our protocol is of constant round complexity and, assuming access to enough cores, computationally bounded only by the number of layers in the circuit to be computed and the block size of a hash function. Using a NVIDIA GPU as our SIMD device, we make several experiments and show that our approach is orders of magnitude more efficient on current *consumer hardware* than any other protocol based on garbled circuits. Finally, we show that this approach is the fastest yet documented assuming a "practical", yet malicious, setting.[2]

Notation. We let $\|$ denote string concatenation and let $r[i]$ be the i'th element of a string r. We let ℓ be the statistical security parameter and κ be the computational security parameter. In particular we let $H(\cdot)$ denote a hash function with a digest of κ bits (in our implementation this will be 160 bits). We assume that Alice is the circuit *constructor* and Bob is the circuit *evaluator* and we will use their names and roles interchangeably.

Overview. Section 2 introduces the idea of parallel implementations and the overall structure of our computation device of choice; the GPU. In Section 3 we go through the overall structure of our protocol. Later, in Section 4 we go

[1] 1-out-of-2 OT is the protocol where the first party, Alice, gives as input two bit-strings (x_0, x_1), and the second party, Bob, inputs a bit b. Bob learns x_b but gets no information on x_{1-b} and Alice gets no information on b.

[2] We refer to "practical" as either financially feasible for a consumer and/or having a liberal statistical security parameter.

through the ideas used to make our protocol suitable in the SIMD model. Then, in Section 5 we discuss the implementation details and finally in Section 6 we review our results.

2 Background

Parallel Approach. In our approach we assume access to a massive parallel computation device which is capable of executing the same instruction on each processor in parallel, but on different pieces of data. Our protocol does not make any assumption on whether such a device has access to shared memory between the processors, or only access to local memory. This applies completely for write privileges, but also for read privileges with only a constant memory usage penalty.

We decided to implement our protocol using the GPU, the motivation being that GPUs are part of practically all mid- to high-end consumer computers. Furthermore, using the GPU eliminates the security problems from outsourcing the computation to a non-local cluster. Also, assuming access to a local cluster seems to be an unrealistic assumption for general practical applications. Using gaming consoles or multi-cores CPUs might also be an option. However, even the latest and best of these have orders of magnitude processors less than the latest GPUs.

Our implementation is done using the CUDA framework which is an extension to C and C++ that allows using NVIDIA GPUs for general computational tasks. This is done by making CUDA programs. Such a program does not purely run on the GPU. It consists of both general C classes, which run on the CPU, and CUDA classes which run on the GPU since the GPU can not communicate directly with the Operating System (OS).

In order to motivate our specific implementation choices it is necessary to describe a general CUDA enabled GPU: Each GPU consists of several (up to 192) *streaming multiprocessors* (SM), each of these again contains between 8 and 192 *streaming processors* (SP), depending on the architecture of the GPU. Each of the SPs within a given SM always performs the same operations at a given point in time, but on different pieces of data. Furthermore, each of these SMs contains 64 KB of *shared memory* along with a few kilobytes of constant cache, which all of the SPs within the given SM must share. For storage of variables each SM contains 64K 32-bit registers which is shared amongst all the SPs. Thus all the threads being executed by a given SM must share all these resources.

We now introduce some notation and concepts which are used in the general purpose GPU community and which we will also use in this paper; a GPU is called a *device* and the non-GPU parts of a computer is called the *host*. This means that the CPU, RAM, hard drive etc., are part of the host. The code written for the host will be used to interact with the OS, that is, it will do all the IO operations needed by the CUDA program. The host code is also responsible for copying the data to and from the device, along with launching code on the device. Each procedure running on a device without interaction with the host is called a *kernel*. Before launching a kernel the host code should complete all

needed IO and copy all the data needed by the kernel to the device's RAM. The RAM of the device is referred to as *global memory*. After a kernel has terminated the host can copy the results from the global memory of the device to its own memory, before it launches another kernel.

A kernel is more than just a procedure of code, it also contains specifications of how many times in parallel the code should be executed and any type of synchronization needed between the parallel executions. A kernel consists of code which is executed in a *grid*. A grid is a 2-dimensional matrix of *blocks*. Each block is a 3-dimensional matrix of threads. Each thread is executed once and takes up one SP during its execution. When all the threads, of all the blocks in the grid, have been executed the kernel terminates. The threads in each block are executed in *warps*, which is a sequence of 32 threads. Thus, the threads must be partitioned into blocks in multiples of the warp size, and contain no branching. The threads can then be executed completely independently and in arbitrary order.

Furthermore, to achieve the fastest execution time one should *coalescence* the data in global memory. That is, to "sort" the data such that the word thread 1 needs is located next to the word thread 2 needs and so on. This makes it possible to load these 32 words for the warp in one go, thus limiting the usage of bandwidth, and in turn significantly increasing the speed of the program. This advice on memory organisation is also relevant for the data in the shared memory. Finally, it is a well known fact [3] that the bottleneck for most applications of the massive parallelism offered by CUDA is the memory bandwidth, thus it should always be a goal to limit the frequency of which a program access data in the global memory.

Maliciously Secure Garbled Circuits. For completeness we now sketch how a generic garbled circuit is constructed. We are given a Boolean circuit description, C, of the Boolean function we wish to compute, f, from which we construct a garbled circuit, GC. For simplicity we assume that each gate consists of two input wires and one output wire. However, we allow the output wire to split into two or more if the output of a given gate is needed as input to more than one other gate. Each wire in C has a unique label, and we give the corresponding wire in GC the same label. Each wire w has two keys associated, k_w^0 and k_w^1, which are independent uniformly random bitstrings. Here k_w^0 represents the bit 0 and k_w^1 represents the bit 1. If the bit on wire w in C is 0, then the value on wire w in GC will be k_w^0, otherwise it will be k_w^1. Each gate in GC consists of a *garbled computation table*. This table is used to find the correct value of the output wire given the correct keys for the input wires. For a gate of C call the left input wire l, the right input wire r and the output wire o. Assume the functionality of the gate is given by $G(\sigma, \upsilon) = \rho$ where $\sigma, \upsilon, \rho \in \{0,1\}$, then the garbled computation table is a random permutation of the four ciphertexts $C_{\sigma,\upsilon} = \mathrm{E}_{k_l^\sigma}\left(\mathrm{E}_{k_r^\upsilon}\left(k_o^\rho\right)\right) = \mathrm{E}_{k_l^\sigma}\left(\mathrm{E}_{k_r^\upsilon}\left(k_o^{G(\sigma,\upsilon)}\right)\right)$ for all four possible input pairs, (σ, υ), using some symmetric encryption function, $\mathrm{E}_{\mathrm{key}}(\cdot)$. I.e., the entries in the garbled computation table consists of "double encryptions" of the output wire's

values, where the keys for each double encryption corresponds to exactly one combination of the input wires' values. The encryption algorithm is constructed such that given k_l^σ and k_r^υ it is possible to recognize and correctly decrypt $C_{\sigma,\upsilon}$, but it is not possible to learn any information about the remaining three encryptions.

Optimized Garbled Circuits. To determine which entry in the garbled computation table is the correct one to decrypt we use permutation bits [21]. The idea is to associate a *permutation bit*, $\pi_i \in \{0,1\}$, with each wire, i, in GC. The value on i is then defined as $k_i^b \parallel c_i$ where $c_i = \pi_i \oplus b$ with b being the bit wire i should represent. We call c_i the *external value*. The garbled computation table is then

$$\left[c_l,\ c_r\ :\ \mathrm{E}^{Gid\|c_l\|c_r}_{k_l^{b_l},k_r^{b_r}}\left(k_o^{G(b_l,b_r)} \parallel c_o\right)\right]^{1,1}_{l=0,r=0},$$

where $c_l = \pi_l \oplus b_l,\ c_r = \pi_r \oplus b_r$, and $c_o = \pi_o \oplus G(b_l, b_r)$, *sorted* on $c_l \| c_r$. This means that given the keys of the input wires the evaluator can decide which entry he needs to decrypt, without learning anything about the bits the wires represent. The encryption function for the keys in the garbled computation table is defined as follows:

$$\mathrm{E}^s_{k_l,k_r}(k_o) = k_o \oplus \mathrm{KDF}^{|k_o|}(k_l, k_r, s),$$

where $\mathrm{KDF}^{|k_o|}(k_l, k_r, s)$ is a *key derivation function* with an output of $|k_o|$ bits, independent of the two input keys, k_l and k_r in isolation, and which depends on the value of some salt, s. As we assume the ROM, we are able to specify the KDF as follows:

$$\mathrm{KDF}^{|k_o|}(k_l, k_r, s) = \mathrm{H}(k_l \parallel k_r \parallel s)\ .$$

This means that the encryption function essentially can be reduced to a single invocation of a robust hash function with output length κ (assuming $\kappa \geq |k_o|$).

We further include the optimization from [11] which will make it possible to evaluate all the XOR gates in the circuit for "free". Free here means that no garbled computation table needs to be constructed or transmitted. The trick is to have a *global key* Δ, which is a uniformly random bitstring of the same length as the wire keys, and then let $k_i^1 = k_i^0 \oplus \Delta$ for all wires i. Regarding the external values, this implies that $\pi_i \oplus 1 = \pi_i \oplus 0 \oplus 1$. So, in order to compute an XOR gate simply compute XOR of the keys of the two input wires of the gate, that is $k_o \parallel c_o = k_l \oplus k_r \parallel c_l \oplus c_r$. Finally, we also eliminate a row of the garbled computation table using the approach of [17]. The trick is to let one of the output keys be the result of the KDF on one input key pair. This key pair is the one where the external values are 0, i.e., $c_l = 0$ and $c_r = 0$. I.e.:

$$k_o^{G(\pi_l\oplus 0,\pi_r\oplus 0)} \parallel c_o = \mathrm{KDF}^{\kappa+1}\left(k_l^{\pi_l\oplus 0},\ k_r^{\pi_r\oplus 0},\ Gid \parallel 0 \parallel 0\right).$$

Depending on the type of gate, this again uniquely specifies the permutation bit of the output wire as $c_o = \pi_o \oplus G(\pi_l \oplus 0, \pi_r \oplus 0)$. The other output key is given

using Δ. The three remaining entries in the garbled computation table are then the appropriate encryptions of these two output keys.

Optimized Approaches to Cut-and-Choose Malicious Security. In general OT is an expensive primitive, and if the evaluator has a large input to the circuit this can contribute significantly to the execution time of the whole protocol. However, the amount of "actual" OTs we need to complete can be significantly reduced by using an *OT extension*: Beaver showed in [1] that given a number of OTs it is possible to "extend" these to give a polynomial number of random OTs which can easily be changed to specific OTs. Thus, making it possible to do a few OTs once, and extend these almost indefinitely. The idea of an OT extension has been optimized even further in [8] and [18] to yield significant practical advantages. Our protocol uses a slightly modified version of the OT extension presented in [18].

The cut-and-choose approach in itself is unfortunately not enough to make a semi honestly secure protocol maliciously secure. In fact, several problems arise from using cut-and-choose to get security against a malicious adversary, these problems can be categorized as follows:

1. "Consistency of input bits"; both parties need to use the same input in all the cut-and-choose instances to ensure that the majority of the garbled circuit evaluations are consistent and that a corrupt evaluator does not learn the output of the function on different inputs.
2. "Selective failure attack"; we must make sure that both the keys the constructor inputs in the OT phase are correct, to avoid giving away a particular bit values of the evaluator's input, depending on failure or not of the evaluation.

Letting $|x|$ be the size of the constructor's input and ℓ the statistical security parameter then the first problem can be solved using $O(|x| \cdot \ell^2)$ commitments to verify consistency in all possible cut-and-choose cases [13]. A more efficient approach is to construct a Diffie-Hellman pseudo random synthesizer, which limits the complexity to $O(|x| \cdot \ell)$ symmetric and asymmetric operations and also solves the selective failure attack [15]. Yet another solution is based on claw-free functions [23].

The selective failure problem can also be solved using different techniques. In [13] it is shown how to do this using a circuit extension which increases the amount of input bits of the evaluator by a factor ℓ. In [23] the problem is solved using a special version OT, known as *comitting OT*.

Our solution is different; we solve the problem of the consistency in the constructor's input bits by using a circuit extension and the consistency of the evaluator by extending each OT by a factor ℓ using the random oracle. The selective failure attack is handled by a novel combination of the OT extension and the use of the free-XOR approach in the garbled circuit. We use these constructions to achieve parallel scalability.

3 High Level Description

We now describe the overall structure of our protocol. For simplicity we assume that only the evaluator is supposed to receive output from the computation. If we wish to compute a circuit where the constructor should also receive output then the circuit extension approach of [13], or the signed output approach of [23], will work directly in our protocol and be scalable in parallel.

Abstractly our protocol can be described as follows:

1. Given a statistical security parameter, ℓ, such that the probability of a total breakdown is at most $2^{-\ell}$, along with a Boolean circuit C, the constructor extends the circuit to get a new circuit, C', that includes a consistency check. Using the description C', the constructor constructs $\ell' = 3.22 \cdot \ell$ GCs in parallel.[3]
2. The constructor then hashes each of the ℓ' GCs along with the keys for the evaluator's output, and sends the digests to the evaluator. These digests makes it possible to avoid sending half of the garbled computation tables as mentioned in [5]. This ends the *garbling phase*.
3. The constructor then sends both of the keys of the evaluator's output wires to the evaluator.
4. The constructor and evaluator engage in OT in order for the evaluator to learn the keys corresponding to his input for all ℓ' circuits. We call this the *OT phase*.
   (a) The constructor and evaluator complete a modified OT extension which is a 1-out-of-2 OTs of random bitstrings.
   (b) For each of these OTs the constructor extends the two random outputs to a $\ell' \cdot \kappa$ "random" bitstring. The first representing the 0-keys of the ℓ' garbled circuits and the other the 1-keys.
   (c) Similarly the evaluator extends his output of each OT to a $\ell' \cdot \kappa$ "random" bitstring, representing either the 0 or 1 keys of the ℓ' garbled circuits depending on his choice in the OT.
   (d) From the circuit generation the constructor will have a 0 and 1 key for each wire in each GC. The constructor then XORs each of the "random" bitstrings she learned from the modified OT extension with the appropriate keys from the circuit generation and sends all these differences to the evaluator.
   (e) The evaluator uses these bitstrings to find the correct input keys for the GCs by a simple XOR operation.
5. The parties then select $\ell'/2$ circuits for verification (using a coin-tossing protocol) and the constructor sends the random seeds used to generate these circuits to the evaluator. We call this and the following three steps for the *cut-and-choose phase*.
6. Using the seeds the evaluator regenerates the circuits' garbled computation tables along with the keys of the output wires and verifies that they are

[3] The constant increase in the amount of GCs stems from the fact that cut-and-choose of ℓ circuits only corresponds to statistical security of $2^{-0.311\ell}$ [15].

correct by hashing them and checking equality with the digests he has already received in Step 2. He also uses the seeds to generate the input keys for the GCs. He uses these keys, the differences he received in the OT phase, along with his outputs from the OT phase, to reconstructs both the 0 and 1 keys and uses these values to verify that the constructor sent the correct differences in the OT phase.

7. After these checks the constructor sends the input keys in correspondence with her input, along with the garbled computation tables of the $\ell'/2$ circuits for which the evaluator was *not* given the seeds.
8. The evaluator then hashes the garbled computation tables of these circuits and verifies them against the hash digests he received in Step 2. He then degarbles the circuits to achieve the output keys along with their respective external values. In the end he then checks consistency of these outputs. We call this the *evaluation phase*.
9. If all checks pass, then the evaluator maps the output keys to their corresponding bits and take the majority of the decrypted outputs of the $\ell'/2$ circuits to be the overall output of the protocol.

4 Specific Details

The Garbled Circuit. First of all, we modify the circuit of the function we wish to compute in order to embed a consistency check for the constructor's input. Assume the function we wish to compute is defined by f as $f(x,y) = (f_1(x,y),\ f_2(x,y))$ with $|x| = \tau_a$, $|y| = \tau_b$ and $f_1(x,y)$ being the (possibly empty) output the constructor is supposed to learn and $f_2(x,y)$ being the output the evaluator is supposed to learn. We now define a new function f' as $f'((x,s),\ (y,r)) = (f_1(x,y),\ (f_2(x,y),\ t))$ where $s \in_R \{0,1\}^\ell$, $r \in_R \{0,1\}^{\tau_a+\ell}$ and $t \in \{0,1\}^\ell$. To compute t define a matrix $\mathbf{M} \in \{0,1\}^{\ell \times \tau_a}$ where the i'th row is the first τ_a bits of $r << i$ where $<<$ denotes the bitwise left shift, i.e. $\mathbf{M}_{i,j} = r[i+j]$. Using this matrix the computation of t is defined as $t = (\mathbf{M} \cdot x) \oplus s$, assuming all binary vectors are in column form.

With this modification the new function computes the same as the original, but requires ℓ extra random bits of input from the constructor and $\tau_a + \ell$ extra random bits from the evaluator. However, the new function returns ℓ extra bits to the evaluator. These ℓ extra bits will work as digest bits and can be used to check that the constructor is consistent with her inputs to the GCs by verifying that they are the same in all the garbled circuits which are evaluated.

This augmentation works since the new function computes, besides the original functionality, a family of universal hash functions where the auxiliary input from both parties defines a particular hash function from this family. The auxiliary output of the augmented function is then the digest of the constructor's input in this universal hash function. The proof that the augmentation is indeed a family of universal hash functions was shown in [16]. Thus this gives statistical security $2^{-\ell}$ when augmenting the function with an ℓ bit digest.

We turn this new function, f', into a circuit description which we then parse. The parsing consists of finding all the gates which can be computed using only

the input wires, calling this set of gates for layer 0. We then find all the gates, not in layer 0, that can be computed using only the input wires and the output wires of the gates in layer 0, calling this layer for layer 1. We continue in this manner until all gates have been assigned a unique layer. The interesting thing to notice here is that we now have a partition of the gates in such a manner that all gates in a single layer can be constructed or evaluated in parallel, in an arbitrary order, only requiring that gates at lower levels have been constructed or evaluated beforehand. Thus, given the keys of the input wires we can construct the garbled computation tables of the gates in layer 0 in an arbitrary order. Moreover, the heavy part of these computations, encryption, can be done in a SIMD manner. The only part of the construction that varies, depending on the type of gate, is which entries in the garbled computation table that should represent a 0-key and which that should represent a 1-key. Notice, however, since we implement the free XOR approach this problem is eliminated, as we can simply multiply the global key with the output of the given gate and always XOR this into the garbled computation table entry which is already representing a 0-key. Still, using the free XOR approach gives another problem, that is the need to further partition each layer into sets of XOR gates and non-XOR gates, in order to achieve complete SIMD *or* to keep the amount of layers and instead execute each layer like it *only* consists of XOR gates *and* execute it like *only* consists of non-XOR gates and only use the relevant result of each of the gates.

Finally, it should be noted that the global key we choose needs to be the same for all the gates in one GC, but different for each of the GCs we make to allow opening in cut-and-choose. Keeping these changes, and this way to parallelize in mind, the protocol for construction is the same as the optimized protocol for generic GC generation previously described, repeated ℓ' times.

The evaluation proceeds in almost the same manner as in the generic garbled circuit evaluation. However, we still use the same paradigm for parallelization as in the construction phase; we degarble each gate in a given layer, in all the $\ell'/2$ circuits, in parallel. Finally, having degarbled all gates, and thus found the keys on the output wires. The evaluator uses the output keys previously received by the constructor to find the bits of his output. The evaluator then checks for a selective failure attack by verifying that each of the ℓ digest bits, on all of the $\ell'/2$ circuits, has the same values. If that is not the case then the evaluator outputs failure. Finally, the evaluator takes the majority of the outputs to be his outputs.

The Modified OT Extension. We use the approach from [18] for the core of our modified OT extension. However, we make a few changes to reduce as many operations as possible to parallel computable hashes of short bitstrings.

Assuming the existence of random oracles and a secure implementation of a κ-bit 1-out-of-2 OT as an ideal resource, the protocol is UC secure against a malicious adversary. For the rest of this section we let τ be the amount of bits in the evaluator's input for the augmented circuit, i.e. $\tau = \tau_b + \tau_a + \ell$.

Define the evaluator's (Bob's) input to the augmented circuit as a bitstring of $y' = y \parallel r$ of τ bits, where y is his original input. Define $\mathrm{H}(\cdot)$ to be a hash function with κ bits output. The modified OT extension goes as follows:

1. Bob chooses $\lceil \frac{8}{3}\kappa \rceil$ pairs of seeds, each consisting of κ random bits. That is, for each $i = 1, \ldots \lceil \frac{8}{3}\kappa \rceil$ let $(l_i^0,\ l_i^1) \in_R \{0,1\}^\kappa \times \{0,1\}^\kappa$ be the i'th seed pair.
2. Alice now samples $\lceil \frac{8}{3}\kappa \rceil$ random bits, $x_1, \ldots, x_{\lceil \frac{8}{3}\kappa \rceil} \in_R \{0,1\}$.
3. Alice and Bob then run $\lceil \frac{8}{3}\kappa \rceil$ OTs where, for $i = 1, \ldots, \lceil \frac{8}{3}\kappa \rceil$, Bob offers $(l_i^0,\ l_i^1)$ and Alice selects x_i, and receives $l_i^{x_i}$.
4. Now, for each of the $i = 1, \ldots, \lceil \frac{8}{3}\kappa \rceil$ pairs of random bits Bob computes the following two vectors of τ bits, using $id_{i,j}$ as a unique ID:

$$L_i^0 = \mathrm{H}\left(id_{i,0} \| l_i^0\right) \| \mathrm{H}\left(id_{i,1} \| l_i^0\right) \| \ldots \| \mathrm{H}\left(id_{i,\tau/\kappa} \parallel l_i^0\right),$$
$$L_i^1 = \mathrm{H}\left(id_{i,0} \| l_i^1\right) \| \mathrm{H}\left(id_{i,1} \| l_i^1\right) \| \ldots \| \mathrm{H}\left(id_{i,\tau/\kappa} \parallel l_i^1\right).$$

5. Now, in the same manner Alice extends each of her outputs of the OT from their original length of κ bits, into strings of τ bits. Thus, Alice computes $L_i^{x_i} = \mathrm{H}\left(id_{i,0} \| l_i^{x_i}\right) \| \mathrm{H}\left(id_{i,1} \| l_i^{x_i}\right) \| \ldots \| \mathrm{H}\left(id_{i,\tau/\kappa} \parallel l_i^{x_i}\right)$.
6. Now, for each $i = 1, \ldots, \lceil \frac{8}{3}\kappa \rceil$ Bob computes a bitstring, $\lambda_i = L_i^0 \oplus L_i^1 \oplus y'$, and sends these to Alice.
7. For each $i = 1, \ldots, \lceil \frac{8}{3}\kappa \rceil$ Alice computes a bitstring as follows

$$L'^{x_i}_i = L_i^{x_i} \oplus (x_i \cdot \lambda_i) = L_i^0 \oplus (x_i \cdot y').$$

8. Alice then picks a uniformly random permutation $\pi : \left\{1, \ldots, \lceil \frac{8}{3}\kappa \rceil\right\} \to \left\{1, \ldots, \lceil \frac{8}{3}\kappa \rceil\right\}$ where, for all i, $\pi(\pi(i)) = i$, and sends these to Bob. Furthermore, define $S(\pi) = \{i | i \leq \pi(i)\}$, that is, for each pair, the smallest index is in $S(\pi)$.
9. Now, for all the $\lfloor \frac{4}{3}\kappa \rfloor$ indexes $i \in S(\pi)$ do the following:
   (a) Alice computes $d_i = x_i \oplus x_{\pi(i)}$ and sends these to Bob.
   (b) Alice and Bob both compute $Z_i = \left(L'^{x_i}_i \oplus L'^{x_{\pi(i)}}_{\pi(i)}\right)$. This is possible for Bob since d_i uniquely determines the way to compute Z_i, i.e. if he should XOR L_i^0 with y'.
10. For all $i \in S(\pi)$, Alice and Bob concatenate Z_i and evaluate equality using the protocol for equality of [18], modified for parallel computation (see the full version of this article), and abort if they are not equal.
11. For each $i = 1, \ldots, \lfloor \frac{4}{3}\kappa \rfloor$ and for each $j = 1, \ldots, \tau$ Alice defines K_j to be the string consisting of the j'th bits from all the strings $L'^{x_i}_i$, i.e. $K_j = L'^{x_1}_1[j] \| L'^{x_2}_2[j] \| \ldots \| L'^{x_{\lfloor \frac{4}{3}\kappa \rfloor}}_{\lfloor \frac{4}{3}\kappa \rfloor}[j]$. This means that she gets τ keys consisting of $\lfloor \frac{4}{3}\kappa \rfloor$ bits.
12. Now, for each $i = 1, \ldots, \lfloor \frac{4}{3}\kappa \rfloor$ and for each $j = 1, \ldots, \tau$ Bob sets M_j to be the string consisting of the j'th bits from all the strings L_i^0, i.e. $M_j = L_2^0[j] \| L_2^0[j] \| \ldots \| L^0_{\lfloor \frac{4}{3}\kappa \rfloor}[j]$.

13. Alice lets Γ_A be the string consisting of all the bits x_i for $i \in S(\pi)$, i.e. $\Gamma_A = x_1 \| x_2 \| \ldots \| x_{\lfloor \frac{4}{3}\kappa \rfloor}$.
14. Bob now computes $Y_j = \mathrm{H}(M_j)$ and achieves $(Y_0, \ldots, Y_\tau)$. He then extends each of these to ℓ' random values. That is, for each $i = 1, \ldots, \ell'$ he computes $Y_j^i = \mathrm{H}\left(id_{i,j} \| Y_j\right)$.
15. Alice computes $X_j^0 = \mathrm{H}(K_j)$ and $X_j^1 = \mathrm{H}(K_j \oplus \Gamma_A)$ and achieves $((X_1^0, X_1^1), \ldots, (X_\tau^0, X_\tau^1))$. She then extends each of these pairs to pairs of ℓ' random values. Specifically for each $i = 1, \ldots, \ell'$ she computes the following:

$$\left(X_j^{0,i}, X_j^{1,i}\right) = \left(\mathrm{H}\left(id_{i,j} \| X_j^0\right), \mathrm{H}\left(id_{i,j} \| X_j^1\right)\right).$$

If the parties have been honest it should be the case, that for each $i = 1, \ldots, \ell'$ and $j = 1, \ldots, \tau$ we have $Y_j^{y'[j],i} = X_j^{y'[j],i}$.

Fitting It Together. After completing the modified OT extension Bob has $\tau \cdot \ell'$ keys of length κ. However, these keys are not consistent with the random keys used for the ℓ' circuits. So, for each of the $\tau \cdot \ell'$ pairs of keys Alice has, she computes the difference between the keys she achieved as a result of the modified OT extension and the actual keys to the given GCs. That, is for each $i = 1, \ldots, \ell'$ and each $j = 1, \ldots, \tau$ she computes $\delta_j^{0,i} = X_j^{0,i} \oplus k_j^{0,i}$ and $\delta_j^{1,i} = X_j^{1,i} \oplus k_j^{1,i}$ where $k_j^{0,i}$ is the 0-key and $k_j^{1,i}$ is the 1-key for the particular wire, j, in the particular GC, i. Alice then sends all the pairs of δs to Bob. For each pair, Bob can only know one X value, that is, either $X_j^{0,i}$ or $X_j^{1,i}$, because of the hiding property of the OT. This means that Bob can compute exactly his choice of key, but not the other. This follows from the security of the free-XOR approach, along with the power of the random oracle for constructing $X_j^{0,i}$ and $X_j^{1,i}$, i.e. they work as one-time-pads for the keys. Thus, we get a linking between the modified OT extension and the GCs.

Finally, Alice also computes a digest of each of her outputs from the OT phase and sends these to Bob. That is, for each $i = 1, \ldots, \ell'$ and each $j = 1, \ldots, \tau$ she computes and sends $\chi_j^{0,i} = \mathrm{H}(X_j^{0,i})$ along with $\chi_j^{1,i} = \mathrm{H}(X_j^{1,i})$ to Bob.

After the cut-and-choose phase Bob will know the following bitstrings for each of his input wires in $\ell'/2$ of the GCs:

- Both the keys for the current input wire, i.e. $k^0,\ k^1 = k^0 \oplus \Delta$.
- Exactly one output of the OT phase, X^b, for his input bit, b, on the current wire.
- Both the difference bitstrings for the current input wire, i.e. δ^0 and δ^1.
- A digest for both the possible outcomes of the OT phase, i.e. $\chi^0 = \mathrm{H}(X^0)$, $\chi^1 = \mathrm{H}(X^1)$.

To verify that δ^0 and δ^1 are correct he computes

$$\delta'^b = k^b \oplus X^b, \quad X'^{\neg b} = \delta^b \oplus \delta^{\neg b} \oplus \Delta, \quad \chi'^{\neg b} = \mathrm{H}(X'^{\neg b}) .$$

He accepts if and only if ${\delta'}^b = \delta^b$ and ${\chi'}^{\neg b} = \chi^{\neg b}$. The intuition of why the check on ${\chi'}^{\neg b}$ is sufficient for the key $k^{\neg b}$ is as follows: If $\delta^{\neg b}$ is incorrect then ${X'}^{\neg b} \neq X^{\neg b}$, in which case, with overwhelming probability, $\mathrm{H}({X'}^{\neg b}) \neq \mathrm{H}(X^{\neg b})$. Now, since Alice does not know which $\ell'/2$ GCs Bob will pick as check circuits, she cannot guess in which of the δ bitstrings she can cheat without being detected. Furthermore, as Bob can check both δ^0 and δ^1, she does not learn anything about his input choices either. In conclusion this little trick prevents a selective failure attack from the constructor.

Parallel Complexity. First see that many of the computationally heavy calculations in the protocol are hashes. Next, notice that these hashes are of "small" bitstrings, bounded by $O(\kappa)$. Now by our approach to parallelization of the garbling and degarbling process we notice that the complexity becomes bounded by the length of the input to the KDF and the depth of the circuit to securely compute. Thus, assuming access to enough parallel processors the garbling and degarbling time will be bounded by $O(\kappa \cdot d)$ where d is the depth of the circuit to garble.

Regarding the modified OT extension notice that all the hashes to be computed in a given step of the modified OT extension can be done independently of each other, and thus in parallel. Looking at these steps from each party's point of view, we see that Step 4 is the step requiring the most computations for Bob. Assume w.l.o.g. that $\tau > \kappa$ then if Bob has access to $p \leq \left\lceil \frac{8}{3}\kappa \right\rceil \cdot \tau$ processors the amount of bits he needs to hash sequentially in the SIMD parallel model is $O(\tau \cdot \kappa^2/p)$. If he has access to more processors then the amount of bits to hash sequentially is only $O(\kappa)$. For Alice the greatest amount of hashes are computed in Step 15. If she has access to $p \leq \tau$ processors then the amount of bits she needs to hash sequentially in said model is $O(\tau \cdot \kappa/p)$. If she has access to more processors, then the amount of bits to hash is only $O(\kappa)$. In conclusion, the overall parallel computational complexity of the protocol is $O(\kappa \cdot d)$, not including the seed OTs.

Finally, note that the communication complexity needed for this protocol is asymptotically the same as for the OT extension described in [18], that is $O(\kappa \cdot (\kappa + \tau)) = O(\kappa \cdot (\kappa + \ell \cdot \tau))$ bits, both for Alice, Bob and in total.

5 Implementation

We now describe how we constructed our implementation in CUDA in order to achieve high efficiency, based on the knowledge of the device hardware and scheduling. It should be noted that we use SHA-1 with 160 bits digest and 512 bits blocks as our hash function.

Garbling. First, notice that we will have a case of SIMD for every circuit in ℓ'. Thus, it is obvious to have each thread in a warp processing a distinct circuit and thus having the blocks be 1-dimensional, consisting of a constant amount of warps. This structure will give us both high block occupancy, and no more than ℓ' threads in each block. We chose to have blocks consist of 32 threads since preliminary tests showed this to be a good choice.

Next we notice that all gates within a single layer can be computed in arbitrary order, thus it is obvious to have one grid dimension be the amount of gates in each layer. Furthermore, as we cannot know which order the blocks will be computed in, we will need to have an iteration of kernel launches, one launch for each layer in the circuit, in order to have the output keys of the previous layer computed and ready for computing the next layer.

Regarding memory management, we first copy the seeds onto the device, and then compute the global keys for all the circuits and the 0 keys for all the input wires in all the circuits, using a unique seed for each circuit. This is done by hashing the seed along with a unique ID in order to get a "random" key (remember we assume the ROM). Afterwards, using the generated keys, we initiate a loop of kernel launches in order to compute each layer of keys and garbled computation table entries in each circuit. Between all these launches, all the currently computed keys, along with the global keys, remain in the global memory of the device so they can be used by the next kernels. Furthermore, we keep all the currently computed garbled computation tables on the device so that all the results can be copied to the host as a batch after all the kernels have finished. In order to save memory we only store the 0-key for each wire, since the 1-key can be efficient computed by simply XORing it with the appropriate global key for a given circuit.

Finally notice that the structure of the kernel for degarbling is the same as for garbling. The only difference is that before the initial launch the garbled computation table for the whole circuit is copied from the host into the global memory along with the initial input keys, one key for each of the 2τ input wires, and a description of the circuit.

Memory Coalescing. We memory coalesced all the data we used, both in the global memory and in the shared memory. As both keys and garbled computation table entries consists of 160 bits (the digest size of SHA-1), i.e. five 32-bit words, we stored all data in *segments* of $32 \cdot 5 = 160$ words. The first entry is the first word of thread 1, the second entry is the first word of thread 2, and so on up to entry 33, which then contains the second word of thread 1, entry 34 contains the second word of thread 2 and so on. Thus, all data access is coalesced in a multiple of the warp size.

The Modified OT Extension. Unlike the generation and evaluation of the GCs, the modified OT extension involves many phases, several of which are depended on the previous phases and results from interacting with the other party. This means that we cannot have a single kernel, or even a single kernel function, in order to complete all the steps of the protocol for each party.

Like we did for the GCs we have coalesced all memory in blocks of 32 words. We also make segments, which consists of $5 \cdot 32 = 160$ words, such that each segment hold a coalesced hash values or a small κ bit data array, for 32 threads. For this reason we again construct kernels to use blocks of 32 threads.

Using this choice, no coalescence conversion needs to be done to use the data from the modified OT extension with our implementation of GCs. Furthermore,

this choice will still keep an efficient and scalable organisation of the memory. Also, as all the data we use for computations here is completely independent, we get the possibility of only launching a single kernel for each step of the protocol in order to avoid kernel launch overhead, resulting from the iterative launching of kernels.

The kernels needed in Step 4 and 5, and Step 14 and 15, are almost the same so we only include a description of Step 4 and 5.

Step 4 and 5. Step 4, involves hashing $2 \cdot \left\lceil \frac{8}{3}\kappa \right\rceil$ seeds τ/κ times. In order to avoid redundant data copying of L_i^0 and L_i^1 to the device when we need to construct λ_i, we compute parts of all the three vectors, L_i^0, L_i^1 and λ_i, in each thread. That is, we include Step 6 in the kernel. To save memory usage and bandwidth we let all the 32 threads of a single block use the same pair of seeds, thus we make each thread in a block compute 160 bits of each of the three vectors L_i^0, L_i^1 and λ_i for the same i. Next, one dimension of the grid is responsible for computing all τ bits of the three vectors, L_i^0, L_i^1 and λ_i, and thus contains $\left\lceil \frac{\tau}{32 \cdot \kappa} \right\rceil$ threads. The other dimension of the grid is responsible for doing this for each of the $\left\lceil \frac{8}{3}\kappa \right\rceil$ vectors that need to be computed. Step 5 proceeds in the same manner, except each block only uses a single seed and each thread only computes a single digest.

6 Experimental Results and Conclusions

For benchmarking our implementation we used the circuit for oblivious 128-bit AES encryption. This circuit is used as benchmark in many previous works including [6,7,15,18]. What makes this circuit a good benchmark is its relatively random structure, its relatively large size, along with its interesting usage for oblivious encryption.

To get the most diverse results we ran our experiments with several different statistical security parameters from 2^{-9} to 2^{-119}. We ran the experiments on two consumer grade desktop computers connected directly by a cross-over cable. At the time of writing each of these machines had a purchase price of less than $1600.

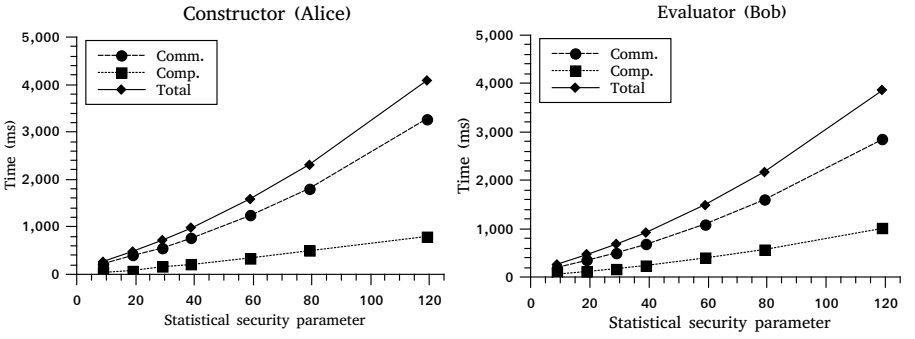

Fig. 1. Timings in milliseconds for both Alice and Bob under different statistical security parameters when computing oblivious 128 bit AES

Table 1. Timing comparison of secure two party computation protocols evaluating oblivious 128 bit AES. d is the depth of the circuit to be computed.

	Security	ℓ	Model	Rounds	Time (s)	Equipment
[7]	Semi honest	-	ROM	$O(1)$	0.20	Desktop
This work	Malicious	2^{-9}	ROM	$O(1)$	0.30	Desktop w. GPU
This work	Malicious	2^{-29}	ROM	$O(1)$	0.83	Desktop w. GPU
[12]	Malicious	2^{-80}	SM	$O(1)$	1.4	Cluster, 512 nodes
[18]	Malicious	2^{-58}	ROM	$O(d)$	1.6	Desktop
This work	Malicious	2^{-59}	ROM	$O(1)$	1.8	Desktop w. GPU
This work	Malicious	2^{-79}	ROM	$O(1)$	2.7	Desktop w. GPU
[12]	Malicious	2^{-80}	SM	$O(1)$	115	Cluster, 1 node

Both machines had similar specifications: an Intel Ivy Bridge i7 3.5 GHz quad-core processor, 8 GB DDR3 RAM, an Intel series-520 180 GB SSD drive, an MSI Z77 motherboard with gigabit LAN and an MSI GPU with an NVIDIA GTX 670 chip and 2 GB GDDR5 RAM. The machines ran the latest version (at the time) of Linux Mint with all updates installed. The experiments were repeated 30 times each and no front end applications were running on either of the machines. These results are summarized in Table. 2 and visualized in Fig. 1. These timings include every aspect of the protocol including loading circuit description and randomness along with communication between the host and device and communication between the parties. However, in the same manner as done in [18] the timing of seed OTs have not been included as this is a computation that practically only is needed once between two parties and thus will get amortized out in a practical context. From these timings we see that the bottleneck of the protocol is the communication complexity. This becomes increasingly obvious the higher the statistical security parameter is.

We believe that our protocol approach along with the implementation yield the best practical results for maliciously secure two-party computation. This is so since the faster timings of [12] is achieved using a large grid with an estimated purchase price of at least $129,168 per party[4] which might not be feasible in the majority of use cases. It should further be noted that their only timings are for statistical security 2^{-80} and that we do not expect a lower security parameter to yield a significant increase in speed due to their approach in parallelization which uses one core per garbled circuit. I.e. they would not be able to utilize more than 28 or 94 cores per player if using statistical security 2^{-9} respectively 2^{-29}. Thus using a less conservative statistical security parameter it seems highly plausible that our protocol implementation will match the pricey grid computer implementation of [12].

Next notice that the approach of [18] achieves a slightly faster result for a conservative statistical security parameter. However, their round complexity is asymptotically greater than ours which could yield performance issues if the protocol were to be executed on the Internet since several packet transmission

[4] Price estimate of a Sun Blade X3-2B with 256 nodes.

Table 2. Timing in milliseconds when computing oblivious 128 bit AES under different statistical security parameters. Communication is on LAN using a cross-over cable.

ℓ	**9**		**19**		**59**	
	Alice	Bob	Alice	Bob	Alice	Bob
IO	4.29 ± 0.0370	4.83 ± 0.357	4.56 ± 0.0290	5.10 ± 0.477	5.75 ± 0.00432	6.35 ± 0.573
OT (total)	37.1 ± 8.48	24.4 ± 5.86	39.0 ± 9.33	24.6 ± 6.01	42.9 ± 9.27	24.2 ± 5.61
OT (comm.)	31.5 ± 8.47	17.3 ± 5.92	32.6 ± 9.32	17.4 ± 5.92	32.8 ± 9.29	15.9 ± 5.66
OT (comp.)	5.55 ± 0.154	7.05 ± 0.408	6.40 ± 0.0468	7.18 ± 0.383	10.1 ± 0.371	8.35 ± 0.317
GC (total.)	230 ± 0.844	235 ± 6.10	441 ± 1.44	434 ± 6.14	1543 ± 5.81	1466 ± 7.36
GC (comm.)	194 ± 0.704	182 ± 6.06	366 ± 2.52	327 ± 6.06	1207 ± 3.25	1080 ± 6.76
GC (comp.)	35.7 ± 0.376	53.2 ± 0.626	75.1 ± 2.36	107 ± 0.732	336 ± 3.45	386 ± 3.32
Total	271 ± 8.38	265 ± 8.27	484 ± 9.55	464 ± 9.62	1591 ± 10.9	1497 ± 9.81
(execution)	300		539		1833	

must be initialized several times during the execution. Furthermore, their timings are based on amortization of 54 instances (or 27 if one is happy with statistical security 2^{-55}). Finally, by an artifact of their approach choosing a lower security parameter will not give significant performance improvements. In particular, a factor 2 in execution time seems to be the absolute maximal time improvement possible by an arbitrary reduction of the statistical security.

In conclusion, we have showed that the construction of a parallel protocol for 2PC in the SIMD parallel model with implementation on the GPU can yield very positive results.

Acknowledgment. The authors would like to thank Benny Pinkas, Thomas Schneider, Nigel P. Smart and Stephen C. Williams for supplying the base circuit which we augmented for our implementation and Roberto Trifiletti for supplying the code we used for circuit parsing.

References

1. Beaver, D.: Correlated pseudorandomness and the complexity of private computations. In: STOC 1996, pp. 479–488. ACM (1996)
2. Bogetoft, P., et al.: Secure multiparty computation goes live. In: Dingledine, R., Golle, P. (eds.) FC 2009. LNCS, vol. 5628, pp. 325–343. Springer, Heidelberg (2009)
3. Nvidia Corporation. NVIDIA CUDA C Programming Best Practices Guide. Technical report (2012)

4. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC 1987, pp. 218–229. ACM (1987)
5. Goyal, V., Mohassel, P., Smith, A.: Efficient two party and multi party computation against covert adversaries. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 289–306. Springer, Heidelberg (2008)
6. Henecka, W., Kögl, S., Sadeghi, A.-R., Schneider, T., Wehrenberg, I.: Tasty: tool for automating secure two-party computations. In: ACM Conference on Computer and Communications Security, pp. 451–462. ACM (2010)
7. Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: USENIX Security Symposium (2011)
8. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 145–161. Springer, Heidelberg (2003)
9. Ishai, Y., Prabhakaran, M., Sahai, A.: Founding cryptography on oblivious transfer – efficiently. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 572–591. Springer, Heidelberg (2008)
10. Kolesnikov, V., Sadeghi, A.-R., Schneider, T.: From dust to dawn: Practically efficient two-party secure function evaluation protocols and their modular design. Cryptology ePrint Archive, Report 2010/079 (2010), `http://eprint.iacr.org/`
11. Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free XOR gates and applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (2008)
12. Kreuter, B., Shelat, A., Shen, C.-H.: Billion-gate secure computation with malicious adversaries. In: 21th USENIX Conference on Security Symposium, p. 14. USENIX (2012)
13. Lindell, Y., Pinkas, B.: An efficient protocol for secure two-party computation in the presence of malicious adversaries. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 52–78. Springer, Heidelberg (2007)
14. Lindell, Y., Pinkas, B.: A proof of security of yao's protocol for two-party computation. J. Cryptology 22(2), 161–188 (2009)
15. Lindell, Y., Pinkas, B.: Secure two-party computation via cut-and-choose oblivious transfer. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 329–346. Springer, Heidelberg (2011)
16. Mansour, Y., Nisan, N., Tiwari, P.: The computational complexity of universal hashing. In: Structure in Complexity Theory Conference, p. 90. IEEE (1990)
17. Naor, M., Pinkas, B., Sumner, R.: Privacy preserving auctions and mechanism design. In: ACM Conference on Electronic Commerce, pp. 129–139. ACM (1999)
18. Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Safavi-Naini, R. (ed.) CRYPTO 2012. LNCS, vol. 7417, pp. 681–700. Springer, Heidelberg (2012)
19. Nielsen, J.B., Orlandi, C.: LEGO for two-party secure computation. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 368–386. Springer, Heidelberg (2009)
20. Nishikawa, N., Iwai, K., Kurokawa, T.: High-performance symmetric block ciphers on multicore CPU and GPUs. International Journal of Networking and Computing 2(2) (2012)
21. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 250–267. Springer, Heidelberg (2009)

22. Pu, S., Duan, P., Liu, J.-C.: Fastplay-a parallelization model and implementation of SMC on cuda based GPU cluster architecture. IACR Cryptology ePrint Archive, 2011:97 (2011)
23. Shelat, A., Shen, C.-H.: Two-Output Secure Computation with Malicious Adversaries. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 386–405. Springer, Heidelberg (2011)
24. Xu, L., Lin, D., Zou, J.: ECDLP on GPU. IACR Cryptology ePrint Archive, 2011:146 (2011)
25. Yao, A.C.: Protocols for secure computations. In: FOCS 1982, pp. 160–164. IEEE (1982)