Fast and Maliciously Secure Two-Party Computation Using the GPU
(Full version)

Tore Kasper Frederiksen[1] and Jesper Buus Nielsen[1]

Department of Computer Science, Aarhus University
jot2re@cs.au.dk, 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 and to learn some output.

This area was introduced in 1982 by Andrew Yao [31], 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 [8]. Unfortunately this protocol was very inefficient as it depended on asymmetric operations for each Boolean gate in the circuit describing the function to compute. However, much research has since been done in the area of 2PC, resulting in practically efficient protocols secure both against both semi honest and malicious adversaries [11, 19, 21, 24, 25, 27].

In general, the protocols secure against a semi honest adversary can become secure against a malicious adversary by adding an additional layer of security. This can be either by compiling a semi honest secure protocol to a maliciously secure protocol [8] or using the cut-and-choose approach where several instances of a semi honest secure protocol is executed in parallel with some random instances being completely revealed to verify that the other party has behaved honestly. However, novel approaches to achieve malicious security do exist, such as the idea of MPC-in-the-head from [14, 18] or by embedding the cut-and-choose part at a lower level of the protocol as done in [25] or [24]. However, assuming access to a large computer grid and using the cut-and-choose approach in a parallel manner has yielded slightly better results than [24] as described in [17].

Motivation. The area of 2PC and multi-party computation, MPC, (when more than two parties supply input) is very interesting as efficient solutions yield several 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. Several other applications for 2PC and MPC includes voting, anonymous identification, privacy preserving database queries etc. For this reason we believe that it is highly relevant to find practically efficient protocols for 2PC and MPC. Most previous approaches have focused on doing this in a sequential model [19, 21, 24]. However, considering the recent evolution of processors we see that the speed of a processor seems to converge,

* 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.

whereas the amount of cores in a processor seems to increase. This in turn implies that the increase in processing power in the future will come from having many cores working in parallel. Thus, constructing both algorithms and cryptographic protocols that work well in a parallel model will be paramount for hardware based efficiency increases in the future. For this reason we have chosen to take a parallel approach to increase the practical speed of 2PC. Previous work taking the parallel approach for efficient implementations of MPC starts with [28] where a cluster of either CPUs or GPUs is used to execute 3072 semi honest protocols for 1-out-of-2 oblivious transfer (OT) followed by gate garbling/degarbling (both based on ECC) in parallel[1]. In [17] the authors use up to 512 cores of the Ranger cluster in the Texas Advanced Computing Center to do OTs along circuit garbling/degarbling in parallel to achieve malicious security using the cut-and-choose approach. In this manner they manage to use the inherit parallelism of the cut-and-choose approach to achieve very fast and maliciously secure 2PCs. Any other work taking a parallel approach to cryptography that we know of focuses either on attacks [30] or simultaneous applications of more primitive cryptographic computations [26].

Security Model. In this paper we focus only on the malicious security model, which is the model providing the greatest (and in our opinion most realistic) security guarantees, but at the price of a greater computational need. We construct and implement a parallel protocol for 2PC secure against a malicious adversary in the UC hybrid model [3], assuming access to a maliciously secure OT e.g. [6] and the existence of random oracles. Our protocol relies solely on symmetric primitives, except for a few "seed" OTs which only need to be done once for each pair of parties.

Technical Approach. Our protocol uses garbled circuits with a cut-and-choose approach in a parallel manner along with a few novel tricks in order to obtain a highly efficient parallel protocol. We manifest and implement this protocol using the massive Same Instruction, Multiple Data (SIMD) parallel power of a consumer-grade GPU in order to achieve the currently fastest practical implementation of maliciously secure 2PC[2].

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 [31] along with the *OT extension* (See Section 2) of [24] 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 we show that this approach is the fastest yet documented assuming a "practical", yet malicious, setting.[3]

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.

[1] 1-out-of-2 OT is the protocol where the first party, Alice, gives as input two bitstrings, and the second party, Bob, gives a single bit. If Bob's bit is 0 then he learns Alice's first bitstring, if it is 1 then he learns Alice's second bitstring. However, Bob never learns more than exactly one of these bitstrings and Alice does not learn anything.

[2] We refer to practically as either financially feasible in computing equipment and/or with a less conservative statistical security parameter.

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

Overview. The rest of the paper is organized as follows: We start in Section 2 with an introduction to the idea of parallel implementations and the overall structure of our computation device of choice; the GPU. Then in Section 3 we go through the overall structure of our protocol. Then in Section 4 we go through the ideas we used to make it suitable for the SIMD model. In Section 5 we discuss the security and complexity of our approach. This is followed by Section 6 where we discuss the implementation details. Then in Section 7 we review our results and end up with a discussion of future work in Section 8.

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. This is in a sense the simplest way of modeling parallel computation, as a device capable of executing distinct instructions on distinct pieces of data is clearly also capable of executing the same instruction on distinct pieces of data. Furthermore, 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.

GPGPU. 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 of outsourcing the computation to a non-local cluster. Also, assuming access to a local cluster seems to be an unrealistic assumption for many practical applications. Finally, using gaming consoles or multi-cores CPUs might also have been an option. However, even the latest and best of these have orders of magnitude cores less than the latest GPUs.

CUDA. 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 cannot 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 [15]: 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 independent and in arbitrary order.

Furthermore, to achieve the fastest execution time one should *coalescence* the data in the 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 [4] 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.

Generic 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.

Now let Alice's input be denoted by x and Bob's input be denoted by y. Then the generic version of semi honestly secure 2PC based on GCs [20] goes as follows:

1. Alice starts by constructing a GC, GC, for a boolean circuit, C, computing the desired function, $f(x, y) = (f_1(x, y),\ f_2(x, y))$.
2. Alice sends to Bob the garbled computation table. She also sends both the keys of the output wires of Bob's output, along with the bit each of these keys represent.
3. Alice now sends keys for the first $|x|$ input wires, corresponding to her desired input. That is, for $i = 1,\ ...,\ |x|$ she sends either $k_{i,a}^0$ or $k_{i,a}^1$ corresponding to her i'th input bit.
4. Next Alice and Bob complete a 1-out-of-2 OT protocol $|y|$ times:
   (a) For $j = 1,\ ...,\ |y|$ Alice obliviously sends the keys $k_{j,b}^0$ and $k_{j,b}^1$ to Bob.
   (b) Bob then chooses exactly one of these keys for each j, without Alice knowing which one.
5. Now Bob has the circuit along with a set of input keys. However, he does not know whether each of the keys to Alice's input represents a 0 or 1 bit, but he does know that the keys for his input represents the correct bits in accordance with his bitstring, y.
6. Bob now degarbles the circuit and learns output keys for all the output wires. He then compares these with the output keys he got from Alice and finds his output bits. He then sends to Alice the keys for the output wires corresponding to her output.
7. Using these keys Alice finds the bits they represent and thus her output.

As soon as any of the players deviate from the protocol, this scheme breaks down completely.

Optimized Garbled Circuits. Several universal techniques for optimizing the generic (both malicious and semi honest security) garbled circuit approach for 2PC exist, which we also implement in our protocol. We now go through the specific and optimized construction of garbled circuit we use.

First notice that we did not specify exactly how to determine which entry in the garbled computation table is the correct one to decrypt given two input wire keys. This is because several different approaches for this exist. However, one efficient approach is the usage of permutation bits along with an efficient key derivation function [27]. The idea is to associate a single *permutation bit*, $\pi_i \in \{0,1\}$, with each wire, i, in the circuit. The value on this wire is then defined as $k_i^b \parallel c_i$ where $c_i = \pi_i \oplus b$ with b being the bit the wire should represent. We call c_i the *external value*. The entries in the garbled computation table are then sorted according to the external values. That is, if the external value on both the left and right wire is 0 then the key in the *first* entry of the table will be encrypted under these two wire keys. If instead the external value on the right wire is 1, then the key in the *second* entry of the table will be encrypted under the left and right wire keys. Thus, we view the external values as a binary number, specifying an entry in the garbled computation table. More formally, entry $c_l,\ c_r$ of the garbled computation table is specified as follows:

$$c_l,\ c_r\ :\ \mathrm{E}^{Gid\|c_l\|c_r}_{k_l^{b_l},k_r^{b_r}}\left(k_o^{G(b_l,b_r)} \parallel c_o\right),$$

where $c_l = \pi_l \oplus b_l,\ c_r = \pi_r \oplus b_r$, and $c_o = \pi_o \oplus G(b_l, b_r)$. This means that given the keys of the input wires the evaluator can decide exactly which entry he needs to decrypt, without learning anything about the bits the input wires represent. Next, see that the encryption function for the keys in the garbled computation tables can be described 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. If we assume the ROM, then 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 can be reduced to a single invocation of a robust hash function with output length κ (assuming $\kappa \geq |k_o|$) along with an XOR operation.

Next we describe the optimization from [16] 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, and no encryption needs to be done in order to evaluate such a gate. The only thing we need to do to get this possibility is to put a constraint on the way the wire keys are constructed. The constraint is very simple, assuming that we wish to construct the keys for wire i, then it must be the case that

$$k_i^1 = k_i^0 \oplus \Delta,$$

where Δ is a *global key*, used in the keys for all wires in the GC. Regarding the external values, this implies the following:

$$\pi_i \oplus 1 = \pi_i \oplus 0 \oplus 1.$$

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, see that a row of the garbled computation table can be eliminated using the approach of [23]. To see this remember that a single Boolean gate takes up four entries of κ bits. However, when using a KDF for encryption we can simply define one of the output keys to be the result of this KDF on one input key pair. This key pair is the one where the external values are 0, i.e. $c_l = 0$ and $c_r = 0$. In short, we must define one of the output keys, and an external value as follows:

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

Depending on the type of gate, this again uniquely specifies the permutation bit of the output wire from this calculation:

$$c_o = \pi_o \oplus G\left(\pi_l \oplus 0, \pi_r \oplus 0\right).$$

The three remaining entries in the garbled computation table are then the appropriate encryptions of the output key or the output key XOR the global difference Δ.

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 [13] and [24] to yield significant practical advantages. Our protocol uses a slightly modified version of the OT extension presented in [24].

Even when using a maliciously secure OT extension the cut-and-choose approach in itself is unfortunately not enough to make a semi honestly secure protocol maliciously secure efficiently. 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 [19]. 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 [21]. Yet another solution is based on claw-free functions [29].

The selective failure problem can also be solved using different techniques. In [19] 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 [29] the problem is solved using a special version OT, known as *committing 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 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 [19], or the signed output approach of [29], 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.[4]

[4] 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 $2^{-0.311\ell}$ [21].

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 [9]. 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 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 correct by hashing them and checking equality with the digests he 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

We now give a more technical description of the protocol and argue why the changes we have made over the "conventional" approaches does not compromise the security.

The Garbled Circuit. We construct the GCs in very much the same manner as described in [31]. However, we use the optimizations for free XOR [16], garbled row reduction [23] along with an efficient encryption function using permutation bits [27]. Our optimization arrives in the way we construct and evaluate the GCs in a scalable parallel manner.

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. However, it must clearly still be the case that honest parties input the same pair (x, s) and (y, r) for each of the ℓ' garbled circuits.

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 [22]. 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 it *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 [24] (see Appendix C), as the base 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
$$\pi : \left\{1, \ldots, \left\lceil \frac{8}{3}\kappa \right\rceil\right\} \to \left\{1, \ldots, \left\lceil \frac{8}{3}\kappa \right\rceil\right\}$$
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 [24], modified for parallel computation (see Appendix B), 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 of course 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 $j = 1,\ ...,\ \tau$ she computes

$$\begin{aligned} \delta_j^{0,i} &= X_j^{0,i} \oplus k_j^{0,i}\ , \\ \delta_j^{1,i} &= X_j^{1,i} \oplus k_j^{1,i} \end{aligned}$$

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 $\chi_j^{0,i} = \mathrm{H}\left(X_j^{0,i}\right)$ along with $\chi_j^{1,i} = \mathrm{H}\left(X_j^{1,i}\right)$ 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

$$\begin{aligned} \delta'^b &= k^b \oplus X^b, \\ X'^{\neg b} &= \delta^b \oplus \delta^{\neg b} \oplus \Delta, \\ \chi'^{\neg b} &= \mathrm{H}\left(X'^{\neg b}\right)\ . \end{aligned}$$

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. However, this approach relies on the same security as the general cut-and-choose approach, and thus we might get some incorrect input keys for our evaluation circuits. Still, the amount of incorrect input keys, making it into the evaluation circuits, is not a majority, except with negligible probability.

5 Security and Complexity

Security of the Modified OT Extension. The overall correctness and security of the modified OT extension follow from the correctness and security of the original OT extension [24]. However, we do make several changes to this protocol. In the following we will specify these change and sketch why they do not compromise the security of the protocol. The most significant changes between the modified OT extension, and the OT extension from [24] are; the non-random construction of y', the use of hashing to construct the strings $L_i^0,\ L_i^1$ and the extension of the results in Step 14 and 15.

Non-random y'. We need y' to be non-random in order for Bob's output of the OT extension to be consistent with his input bits for the GCs. We do this as part of the OT protocol in order to eliminate the need for proving equality of random bits and sending permutation bits which would be the normal approach in order to change random OTs into OTs of specific bitstrings and choices. By embedding this in the extension itself we save some rounds of communication complexity and make the whole OT extension phase simpler by eliminating these post processing steps. Intuitively it does not compromise the security for Bob as y' is one-time-padded with the pseudo random strings L_i^0 in Step 6 and 7. It does not compromise the security for Alice either, because of the bit switching in the end of the protocol (Step 11 and 12), along with the fact, that Alice's vector of $x_1, \ldots, x_{\lceil \frac{8}{3}\kappa \rceil}$ is random, and thus that Bob has no idea if y' will be XORed into a L_i^0 in Step 7.

Construction of the L_i^0s and L_i^1s. In [24] a pseudo random generator is used to extend a random string of length κ to a pseudo random string of length τ. However, our approach is based on invocations of hash functions on a common seed concatenated with a unique ID. This is clearly secure in the random-oracle model.

Step 14 and 15. Like for the L_i^0s and L_i^1s the extensions in Step 14 and 15 does not compromise security because we assume the ROM and so as long as we extend each string along with a unique ID we can assume the output is random.

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 14 is the step requiring the most computations for Bob. If Bob has access to $p \leq \ell' \cdot \tau$ processors the amount of bits he needs to hash sequentially in the SIMD parallel model is $O(\tau \cdot \ell' \cdot \kappa / p)$. 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 $p \leq \tau \cdot \ell'$ processors then the amount of bits she needs to hash sequentially in said model is $O(\tau \cdot \ell' \cdot \kappa / p)$. 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 [24], that is $O(\kappa \cdot (\kappa + \ell \cdot \tau))$ bits, both for Alice, Bob and in total.

6 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 [12] as our hash function.

Gate Generation.

Kernel Structure. 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. Now, since ℓ' can vary but is generally greater than 50, and since preliminary

tests showed that two warps in a block achieved both greater occupancy and higher efficiency than when blocks consisted of a single warp, we chose to have blocks consist of 32 threads. A caveat with this is that if we wish to have ℓ' not being a multiple of 32, we will need to allocate unused memory and cores and thus have SPs sit idle.

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.

Steps 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

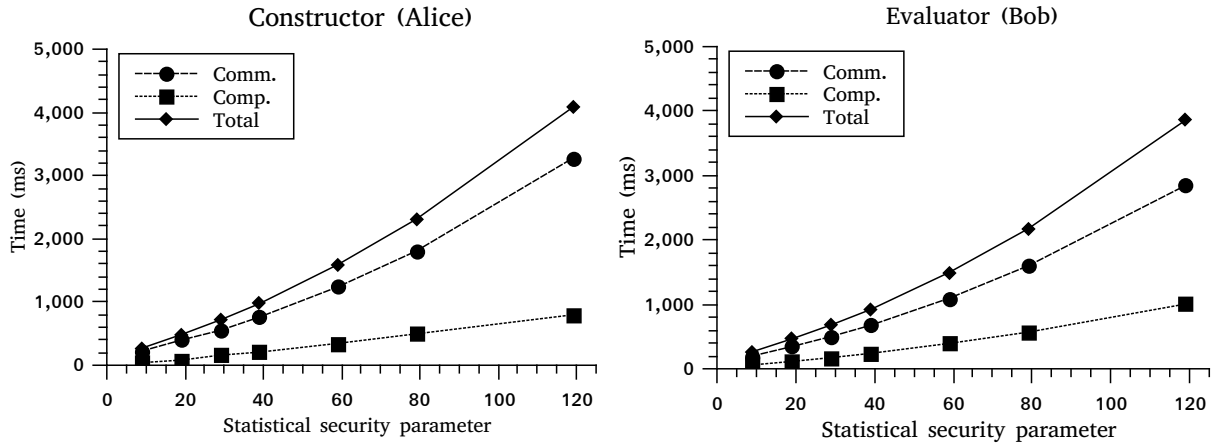

Fig. 1. Timings in milliseconds for both Alice and Bob under different statistical security parameters when computing oblivious 128 bit AES.

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.

Further Improvements. For constructing and evaluating the GCs the hash operations are clearly the main contributing time factor. However, regarding the modified OT extension it turns out, that computing the hash values on the device, barely gave an improvement in the overall execution time, and that the main contributing time factor was that of transposing bits, i.e. Steps 11 and 12. In order to achieve a significant improvement in execution time we need to implement these steps efficiently in parallel. In order to do this we need to keep the overall hardware structure and memory hierarchy in mind.

First of all, we should notice that in order to construct one word of K_j or M_j, we need a single bit from 32 different words in $L'^{x_i}_i$ or L_i^0. In our memory organization, these are located in non-consecutive order. However, it should be noted that the remaining 31 bits of each of the 32 words are needed in the next 31 K bitstrings, K_{j+1}, ..., K_{j+31}. Thus, depending on the caches available, it makes sense to construct the first word of K_j, K_{j+1}, ..., K_{j+31} in a batch. That is, to load 32 words of $L'^{x_i}_i$ or L_i^0 and use a single bit from each of these to construct the first word of K_j, K_{j+1}, ..., K_{j+31}. For this approach to be successful we need a cache of $32 \cdot 32 = 1024$ words, or 4 kilo bytes in a 32 bit system. Fortunately, this is well within the amount of shared memory on a device.

Finally, consider the protocol for parallel equality. It is simple to implement on the device, by again having blocks of 32 threads and a grid of all the blocks needed to compute the individual digests. For each party we start by loading the input string into global memory and then construct a hash value of each, sufficiently large, chunk of bits, by loading the bits directly from global memory and storing the result back in global memory.

Using all these parallel optimizations for the modified OT extension, experiments show that the major contributing time factor is Step 15, as one might also expect, since it involved the largest amount of bits to hash.

7 Experimental Results

Both the theory and implementation of the parallel version for maliciously secure GCs we presented is simple and straight forward, yet very efficient. We show this with the following tests. All of these tests are based on

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
[11]	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
[17]	Malicious	2^{-80}	SM	$O(1)$	1.4	Cluster, 512 nodes
[24]	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
[17]	Malicious	2^{-80}	SM	$O(1)$	115	Cluster, 1 node

the same, commonly used, circuit for oblivious 128 bit AES encryption.[5] This circuit is used as benchmark both in [10, 11, 21, 24], and many more implementations of 2PC for boolean functions. What makes this circuit a good benchmark is its relatively random structure, its relatively large size, along with its very obvious practical usage, i.e. 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. 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 [24] 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.

8 Conclusion

We believe that our protocol approach along with the implementation yields the best practical results for malicious security two-party computation. This is so since the faster timings of [17] is achieved using a large grid with an estimated purchase price of at least $129,168 per party[6] 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 [17].

[5] We thank Benny Pinkas, Thomas Schneider, Nigel P. Smart and Stephen C. Williams for supplying the base circuit which we augmented for our implementation.

[6] We contacted the authors of [17] who unfortunately did not have any price estimate on the Sun Blade x6240 system which they used for their timings results. Furthermore, as Sun Blade x6240 has reached end-of-life the estimate is based on the minimal price of a 256 core x86 system of the current successor Sun Blade x6240, i.e. the Sun Blade X3-2B.

Next notice that the approach of [24] 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 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 PRAM model with implementation on the GPU can yield very positive results.

Future Work. Even though the time needed for the seed OTs can be amortized out from repeated use of the protocol it would still be interesting to see how fast these could be done, in particular, in parallel using the GPU.

Other interesting aspects one could try out with this protocol is to use another key derivation function, such as one based on AES. Furthermore, using our parallel protocol for covert security would also be interesting, since most of the communication complexity can be eliminated in the covert model when using seeds for generations of the garbled circuits.

Finally, implementing some working set mechanism (as done in [17]) would be interesting, as it would make it possible to garble extremely large circuits.

The Code. The rough implementation we did for this work is freely available for non-commercial use at `http://daimi.au.dk/~jot2re/cuda`.

Acknowledgment. The authors would like to thank Roberto Trifiletti for supplying the code we used for circuit parsing and Springer for publishing the extended abstract of this work [7].

References

1. Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, STOC '96, pages 479–488, New York, NY, USA, 1996. ACM.
2. Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *Financial Cryptography*, pages 325–343, 2009.
3. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. *Foundations of Computer Science, IEEE Annual Symposium on*, 0:136, 2001.
4. Nvidia Corporation. NVIDIA CUDA C Programming Best Practices Guide. Technical report, October 2012.
5. Ivan Damgård. A design principle for hash functions. In *CRYPTO*, pages 416–427, 1989.
6. Ivan Damgård, Jesper Buus Nielsen, and Claudio Orlandi. Essentially optimal universally composable oblivious transfer. In Pil Joong Lee and Jung Hee Cheon, editors, *ICISC*, volume 5461 of *Lecture Notes in Computer Science*, pages 318–335. Springer, 2008.
7. Tore Kasper Frederiksen and Jesper Buus Nielsen. Fast and maliciously secure two-party computation using the gpu. In *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, June 25-28, 2013. Proceedings*, LNCS. Springer, 2013.
8. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.
9. Vipul Goyal, Payman Mohassel, and Adam Smith. Efficient two party and multi party computation against covert adversaries. In *EUROCRYPT*, pages 289–306, 2008.
10. Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 451–462. ACM, 2010.
11. Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.

12. National institute˜of˜standards˜and technology. FIPS 180-2, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-2. Technical report, DEPARTMENT OF COMMERCE, August 2002.
13. Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *In CRYPTO 2003, Springer-Verlag (LNCS 2729*, pages 145–161. SpringerVerlag, 2003.
14. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In *CRYPTO*, pages 572–591, 2008.
15. David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*. Morgan Kaufmann, 1 edition, February 2010.
16. Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.
17. Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.
18. Yehuda Lindell, Eli Oxman, and Benny Pinkas. The ips compiler: Optimizations, variants and concrete efficiency. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 259–276. Springer, 2011.
19. Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Proceedings of the 26th annual international conference on Advances in Cryptology*, EUROCRYPT '07, pages 52–78, Berlin, Heidelberg, 2007. Springer-Verlag.
20. Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.
21. Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *TCC*, volume 6597 of *Lecture Notes in Computer Science*, pages 329–346. Springer, 2011.
22. Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. In *Structure in Complexity Theory Conference*, page 90. IEEE Computer Society, 1990.
23. Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *ACM Conference on Electronic Commerce*, pages 129–139, 1999.
24. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700. Springer, 2012.
25. Jesper Buus Nielsen and Claudio Orlandi. Lego for two-party secure computation. In *TCC*, pages 368–386, 2009.
26. Naoki Nishikawa, Keisuke Iwai, and Takakazu Kurokawa. High-performance symmetric block ciphers on multicore cpu and gpus. *International Journal of Networking and Computing*, 2(2), 2012.
27. Benny Pinkas, Thomas Schneider, Nigel Smart, and Stephen Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-10366-7_15.
28. Shi Pu, Pu Duan, and Jyh-Charn Liu. Fastplay-a parallelization model and implementation of smc on cuda based gpu cluster architecture. *IACR Cryptology ePrint Archive*, 2011:97, 2011.
29. Abhi Shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 386–405. Springer, 2011.
30. Lei Xu, Dongdai Lin, and Jing Zou. Ecdlp on gpu. *IACR Cryptology ePrint Archive*, 2011:146, 2011.
31. Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

A Detailed Benchmark

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**		**29**		**39**	
	Alice	Bob	Alice	Bob	Alice	Bob	Alice	Bob
IO	4.293 ± 0.03699	4.833 ± 0.35709	4.558 ± 0.0290	5.104 ± 0.4773	4.888 ± 0.0418	5.442 ± 0.49490	5.189 ± 0.033891	5.737 ± 0.53749
OT (total)	37.1 ± 8.483	24.4 ± 5.855	39.0 ± 9.325	24.6 ± 6.012	38.0 ± 8.669	26.3 ± 6.180	36.4 ± 7.655	26.7 ± 6.195
OT (comm.)	31.5 ± 8.467	17.3 ± 5.915	32.6 ± 9.320	17.4 ± 5.918	30.5 ± 8.653	18.7 ± 6.238	28.0 ± 7.695	18.8 ± 6.258
OT (comp.)	5.55 ± 0.1543	7.05 ± 0.408	6.40 ± 0.0468	7.18 ± 0.383	7.57 ± 0.1011	7.6 ± 0.265	8.4 ± 0.120	7.9 ± 0.264
Circuits (total.)	229.8 ± 0.844	235.3 ± 6.103	441 ± 1.441	434 ± 6.136	674 ± 1.667	649 ± 5.306	933 ± 3.216	887 ± 4.435
Circuits (comm.)	194.1 ± 0.704	182.1 ± 6.056	366 ± 2.523	327 ± 6.055	522 ± 0.954	483 ± 5.290	734 ± 3.558	656 ± 4.307
Circuits (comp.)	35.7 ± 0.376	53.2 ± 0.626	75.1 ± 2.363	107.0 ± 0.732	152.4 ± 1.460	166.0 ± 0.807	198.9 ± 3.578	231 ± 1.170
Total	271.1 ± 8.384	264.5 ± 8.274	484 ± 9.552	464 ± 9.621	717 ± 8.665	681 ± 8.561	975 ± 8.448	920 ± 7.761
(Protocol)	300.2		539.0		833.1		1118	

ℓ	**59**		**79**		**119**	
	Alice	Bob	Alice	Bob	Alice	Bob
IO	5.750 ± 0.00432	6.350 ± 0.57286	6.483 ± 0.1677	6.807 ± 0.2740	7.540 ± 0.0051	7.994 ± 0.16094
OT (total)	42.9 ± 9.268	24.2 ± 5.606	42.1 ± 7.697	27.3 ± 6.176	42.2 ± 6.829	21.3 ± 4.367
OT (comm.)	32.8 ± 9.292	15.9 ± 5.661	30.2 ± 7.697	18.5 ± 6.249	27.7 ± 6.940	11.9 ± 4.265
OT (comp.)	10.08 ± 0.3707	8.35 ± 0.317	11.85 ± 0.3159	8.75 ± 0.327	14.48 ± 0.5550	9.4 ± 0.508
Circuits (total.)	1542.8± 5.806	1466.2± 7.356	2259 ± 3.937	2139 ± 5.540	4029 ± 5.433	3831 ± 7.732
Circuits (comm.)	1206.7± 3.247	1080.1± 6.764	1775 ± 2.907	1579 ± 5.575	3246 ± 3.421	2835 ± 6.941
Circuits (comp.)	336.1 ± 3.445	386.1 ± 3.322	484.5 ± 4.221	559.8 ± 2.641	783.0 ± 3.388	995.8 ± 2.779
Total	1591.4± 10.911	1496.8± 9.812	2308 ± 9.738	2173 ± 7.991	4079 ± 8.652	3860 ± 7.942
(Protocol)	1832.9		2657.6		4643.2	

B Parallel Equality

We describe a simple secure protocol for evaluating equality on two strings which leaks both strings if they differ. This protocol is the same as the one described in [24] except that this one uses a hash function that

computes many small hashes in parallel and then combines these to a single hash value instead of a sequential hash function.

Define Alice's input to be a bitstring, x, and Bob's input to be a bitstring, y. Furthermore, let $\mathrm{H}(\cdot)$ be a cryptographically secure hash function which has a digest of κ bits and a block size of ρ bits. The protocol then proceeds as follows:

1. Alice chooses a random string $r \in_R \{0,1\}^\kappa$.
2. She then views her input string and r as $\left\lceil \frac{|x|+\kappa}{\rho} \right\rceil$ blocks, each of ρ bits. In parallel, she then hashes each of these blocks using the hash function $\mathrm{H}(\cdot)$.
3. She now has $\left\lceil \frac{|x|+\kappa}{\rho} \right\rceil$ hash values. She then concatenates two adjacent digests, and hash all of these.
4. She continues in this manner, recursively concatenating and hashing the results from the previous round. At the end she has a single hash value, call it c, which she sends to Bob.
5. Bob then sends y to Alice.
6. Alice sends $x,\ r$ to Bob and checks locally that $x = y$.
7. Bob also views x concatenated with r as a string of ρ bit blocks, which he, like Alice, recursively hashes and concatenates to achieve a single hash value, c'.
8. Bob then checks if $c' = c$ and that $x = y$.
9. If all checks are successful then the strings are equal, otherwise they are not.

Notice that if the original, sequential, hash function is "good", then this block wise recursive hashing will also result in a hash digest that is "good". This is a result due to Damgaard [5].

C The OT Extension of [24]

The following protocol from [24] shows how we can get a practically unbounded amount of 1-out-of-2 κ bits OTs using only $\lceil \frac{8}{3}\kappa \rceil$ seed OTs of κ bit strings. The extension goes as follows, using ψ to denote the amount of OTs we wish to construct with P_2 denoting the player giving the input and P_1 denoting the player choosing the output:

1. P_1 samples $\Gamma_B \in_R \{0,1\}^\psi$ and for $i = 1,\ ...,\ \lceil \frac{8}{3}\kappa \rceil$, it also samples $L_i^0,\ L_i^1 \in_R \{0,1\}^\kappa$.
2. P_2 samples $y_1,\ ...,\ y_{\lceil \frac{8}{3}\kappa \rceil} \in_R \{0,1\}$.
3. P_1 and P_2 then run $\lceil \frac{8}{3}\kappa \rceil$ instances of a 1-out-of-2 OT protocol in the following manner:
   - For $i = 1,\ ...,\ \lceil \frac{8}{3}\kappa \rceil$ P_1 offers $(L_i^0,\ L_i^1)$ to P_2.
   - P_2 chooses $y_i \in \{0,1\}$ such that it receives $L_i^{y_i}$.
   - Now, P_1 runs a pseudo random number generator on all the candidate value, $\left(L_i^0,\ L_i^1\right)$:
   $$Y'^0_i = \mathrm{prg}^\psi\left(L_i^0\right) \quad , \quad Y'^1_i = \mathrm{prg}^\psi\left(L_i^1\right).$$
   - P_2 now runs the same pseudo random number generator on all the $\lceil \frac{8}{3}\kappa \rceil$ values of $L_i^{y_i}$ to extend each of these from their original length of κ bits into strings of ψ bits. Thus it learns $Y'^{y_i}_i = \mathrm{prg}^\psi\left(L_i^{y_i}\right)$.
   - Now, P_1 computes $\lceil \frac{8}{3}\kappa \rceil$ bitstrings, $\lambda_1,\ ...,\ \lambda_{\lceil \frac{8}{3}\kappa \rceil}$, and sends these to P_2:
   $$\lambda_i = Y'^0_i \oplus Y'^1_i \oplus \Gamma_B.$$
   - For $i = 1,\ ...,\ \lceil \frac{8}{3}\kappa \rceil$ P_2 computes a semi authenticated bit as follows
   $$|y_i\rangle = \left(y_i,\ Y'^{y_i}_i \oplus (y_i \cdot \lambda_i)\right) = \left(y_i,\ Y'^0_i \oplus (y_i \cdot \Gamma_B)\right)$$
4. P_2 now picks a uniformly random permutation:
   $$\pi\ :\ \left\{1,\ ...,\ \left\lceil \frac{8}{3}\kappa \right\rceil\right\} \to \left\{1,\ ...,\ \left\lceil \frac{8}{3}\kappa \right\rceil\right\},$$
   such that for all i; $\pi\left(\pi(i)\right) = i$. That is, the permutation is a pairing. P_2 sends this pairing to P_1. Given this pairing, let $\mathcal{S}(\pi) = \{i | i < \pi(i)\}$. That is, $\mathcal{S}(\pi)$ contains the $\lfloor \frac{4}{3}\kappa \rfloor$ elements which have the smallest index of all pairs in π.

5. Now, for all the $\left\lfloor \frac{4}{3}\kappa \right\rfloor$ indices, $i \in \mathcal{S}(\pi)$, do the following:
   - P_2 announces $d_i = y_i \oplus y_{\pi(i)}$.
   - P_1 and P_2 computes
   $$|z_i\rangle = |y_i\rangle \oplus |y_{\pi(i)}\rangle \oplus d_i.$$
   - Notice that P_1 can simply compute this as:
   $$|z_i\rangle = \left(z_i,\ {Y'}^0_i \oplus {Y'}^0_{\pi(i)} \oplus (d_i \cdot \Gamma_B)\right).$$
   - P_1 and P_2 concatenate all their bitstrings of $|z_i\rangle$ and use a hash function to reduce them to just κ bits. They then use the protocol for *equality* to compare these strings. If they are not equal then they abort the protocol.
6. P_1 then defines x_j be the j'th bit of Γ_B and M_j to be the string consisting of the j'th bits from all the strings ${Y'}^0_i$, i.e. $M_j = {Y'}^0_j[1] \parallel {Y'}^0_j[2] \parallel ... \parallel \left({Y'}^0_j\left[\left\lfloor \frac{4}{3}\kappa \right\rfloor\right]\right)$. So, because of the use of the pseudo random number generator we get ψ bits, x_j, and ψ bitstrings, M_j.
7. P_2 then lets Γ_A be the string consisting of all the bits, y_i, i.e. $\Gamma_A = y_1 \parallel y_2 \parallel ... \parallel y_{\left\lfloor \frac{4}{3}\kappa \right\rfloor}$ and lets K_j be the string consisting of the j'th bits from all the strings ${Y'}^0_i \oplus (y_i \cdot \Gamma_B)$, i.e. $K_j = ({Y'}^0_1 \oplus (y_1 \cdot \Gamma_B))[j] \parallel ({Y'}^0_2 \oplus (y_2 \cdot \Gamma_B))[j] \parallel ... \parallel \left({Y'}^0_{\left\lfloor \frac{4}{3}\kappa \right\rfloor} \oplus \left(y_{\left\lfloor \frac{4}{3}\kappa \right\rfloor} \cdot \Gamma_B\right)\right)[j]$.
8. P_1 now uses the hash function:
   $$Y_j = \mathrm{H}(M_j)$$
   on each of her ψ bitstrings. Thus, he ends up having ψ strings of κ bits, $(Y_1,\ ...,\ Y_\psi)$ along with Γ_B.
9. P_2 also uses the hash function:
   $$X_{i,0} = \mathrm{H}(K_i) \quad , \quad X_{i,1} = \mathrm{H}(K_i \oplus \Gamma_A),$$
   twice on each of her ψ bitstrings, and ends up with ψ pairs of bitstrings, $((X^0_1,\ X^1_1)\ ...,\ (X^0_\psi, X^1_\psi))$.

Now because of the structure of the protocol for LaBit and aBit, it will be the case, for $i = 1,\ ...,\ \psi$, that if $x_i = 0$ then $Y_i = X^0_i$ and if $x_i = 1$ then $Y_i = X^1_i$, i.e. a random OT.