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 secure two-party computation, based on Yao's garbled circuit and an efficient OT extension, in a parallel computational model. The implementation is done using CUDA and yields the fastest results for maliciously secure two-party computation in a realistic and practical setting by using a simple consumer grade CPU and GPU. Our protocol further introduces 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 [30], 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 [10, 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 the do-and-compile approach of [8] where a protocol secure against a semi honest adversary is compiled into a maliciously secure protocol using zero-knowledge proofs, or using the cut-and-choose approach where several instances of a semi honestly 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 [13, 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 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 yields 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 around

* The authors acknowledge support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation, within which this work was performed.

3.0 Ghz, 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 the 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 for MPC start 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 each of the OTs along with each of the circuit garblings 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 [29] 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, secure in the UC hybrid model [3] assuming access to malicious secure OT, e.g. [6], and the Random Oracle Model (RAM). Our protocol relies solely on symmetric primitives, except for a few 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 an actual implementation, along with a general protocol, for efficient maliciously secure 2PC assuming access to a SIMD, or simply Parallel Random Access Model (PRAM), computation device. The protocol is based on Yao's garbled circuit approach [30] along with the OT extension of [24] and a few novel ideas. Our protocol is of constant round complexity and assuming access to enough cores our protocol is computationally bounded only by the asymptotic amount of layers in the circuit to be computed. Furthermore, using a NVIDIA GPU as our SIMD device, we make several experiments and show that our approach is several orders of magnitude more efficient on current consumer hardware than sequential protocols based on the cut-and-choose approach. Finally, we show that this approach is the fastest yet documented (for less conservative statistical security parameters), even compared to [17] which assumes access to a large computation grid.

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. Next, 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

[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 bitstring 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.

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. In order to implement our protocol we have decided to use a specific computation device, that is, the GPU. This choice seems to be the most obvious at this point in time since GPUs are part of practically all mid- to high-end commercial computers, both for personal and processional use, and both laptops and desktops. Furthermore, using the GPU eliminates potential security problems that might arise if one is to outsource the computation to a non-local grid or cluster. Also assuming access to a local grid or cluster seems to be an unrealistic assumption for general practical applications. Finally, using gaming consoles or multi-cores CPUs might also be an option. However, even the latest and fastest of these have orders of magnitude processors less than the latest GPUs. Such as 8 cores on a Cell processor in Sony PlayStation 3, 10 cores on an Intel Xeon Westmere and 2048 cores on an ATI Radeon 7970 or 1536 cores on a NVIDIA GTX 680.

CUDA. Our implementation is implemented using the CUDA framework which is an extension to C and C++. This is a framework used to harvest the massive parallel power of 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 contain code that run on the GPU. This is so, since the GPU can not communicate with the Operating System (OS) in the same manner as the CPU. For example, it is not possible to use any code running on the GPU to do Input/Output (IO) with the OS during computation. Thus code running on the GPU is very much like a batch program; when it has been launched there is no further interaction with it until it has terminated.

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 (the newer, the more cores per SM) [14]. 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. Each of these SMs further contains 64 KB of shared memory, which is a cache all of the SPs within the given SM can share, along with a few kilobytes of constant cache. For storage of variables each SM contains 64K 32-bit registers which is shared amongst all the SPs. Thus all the threads currently being executed by a given SM must share all these resources.

We now introduce some notation and concepts which are used in the GPGPU 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 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 returned, the host can copy the results from the global memory of the device to its own memory, before it launches another kernel.

A kernel consists of 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 consists of a 2- dimensional matrix of *blocks*. Each block then consists of another 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, has been executed, the kernel terminates. The threads in each block are executed in *warps*, which is a sequence of 32 threads. Thus threads must be partition into blocks containing a multiple of the warp size, 32, which contains no branching and which can be executed completely independently and in arbitrary order, to achieve optimal execution speeds.

Furthermore, to achieve the fastest executions 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. If this is the case then these 32 words can be loaded in one go, limiting the usage of bandwidth, and in turn significantly increasing the speed of the program. This advice on memory organization is also relevant for the data in 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 global memory as much as possible.

Maliciously Secure Garbled Circuits.

Generic Garbled Circuits. For completeness we remind the reader how a generic garbled circuit is constructed. We are given a boolean circuit description, C, of the boolean function we wish to compute, f. For simplicity we also assume that the description consists of fan-in 2 and fan-out 1 gates (we denote input and output gates simply as "wires" as they do not do any bit manipulations). Such a gate consists of two input wires and one output wire. 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. We construct a garbled circuit, GC, over the general boolean circuit C. On each gate of GC we call the left input wire k_l, the right input wire k_r and the output wire k_o. Each wire, k_w, has two possible keys; say k_w^0 and k_w^1 which are independent and identically distributed (iid) values. Here k_w^0 represents the bit 0 and k_w^1 represents the bit 1. If the bit on a given wire in C is 0, then the wire k_w in GC, will have value k_w^0, otherwise it will have value 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 of the gate given a specific value on each of the garbled gate's input wires. Assume the functionality of a given gate is defined as $G(\sigma, \tau) = \rho$ with $\sigma,\ \tau,\ \rho \in \{0, 1\}$, then the garbled computation table is a random permutation of $\mathrm{E}_{k_l^\sigma}\left(\mathrm{E}_{k_r^\tau}\left(k_o^\rho\right)\right) = \mathrm{E}_{k_l^\sigma}\left(\mathrm{E}_{k_r^\tau}\left(k_o^{G(\sigma,\tau)}\right)\right)$ for all four possible input pairs, $(\sigma,\ \tau)$, using some symmetric encryption function, $\mathrm{E}_{\mathrm{key}}(\cdot)$. We further add two constraints on the garbled computation tables:

- Given the two input wire keys, k_l^σ and k_r^τ, it should only be possible to correctly decrypt the "correct" of the four entries in the garbled computation table.
- It should further be possible to know which of the four entries is the correct one to decrypt.

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. To remedy this, two general approaches exist; using the GMW compiler [8] to add zero-knowledge proof for the steps in order to "force" honest execution from each of the parties, or using the cut-and-choose approach [19] which involves executing the semi honest protocol several times in parallel, and then choose a sample of the executions to verify that the other player has behaved honestly (which also leads to other security problems that must be fixed). Much research have been made on the compiler and cut-and-choose approach, yielding good results, especially for the cut-and-choose approach [19, 20, 24, 25].

Optimized Generic Garbled Circuits. Several universal techniques for optimizing the generic (both malicious and semi honest securety) garbled circuit approach for 2PC exist, which we of course 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:

$$c_l,\ c_r\ :\ \mathrm{E}^{Gid\|c_l\|c_0}_{k_l^{b_l},k_r^{b_r}}\left(k_o^{G(b_l,b_r)}\parallel c_o\right),$$

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.

We further include 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 other output key is chosen pseudo randomly. 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. When using the cut-and-choose approach to achieve malicious security on garbled circuits we will need to do ℓ OTs for each of the evaluator's input bits. 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 was possible to "extend" these to give an arbitrary number of random OTs which could then easily be changed to specific OTs. Thus making possible to do a few OTs once, and then continue to extend these each time the parties wish to do a secure computation with each other. The idea of an OT extension has been optimized even further in [12] and [24] to yield significant practical advantages. We use this approach in our protocol using a slightly modified version of the extension presented in [24].

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 the cut-and-choose to get security against a malicious adversary, these problems can be categorized as follows:

- Consistency in input bits; both parties need to use the same input in all the ℓ instances that is repeated to ensure that a corrupt party does not learn the output of the function on different inputs.
- Selective failure attack; we must make sure that the keys the constructor inputs in the OT phase are correct, to avoid giving away particular bit values of the evaluator's input.

The first problem can be "trivially" solved by making $O(|x| \cdot \ell^2)$ commitments to be able to verify consistency in all possible cut-and-choose choices [19]. A more efficient approach consists of constructing a Diffie-Hellman pseudo random synthesizer, which limits the complexity of this step to $O(|x| \cdot \ell)$ symmetric operations along with a "batch OT" operation [21], which construction also solves the selective failure problem. Another solution to the selective failure problem consists of an extension to the circuit to be computed [19].

Our solution is very different, in that we solve the problem of the consistency in the constructor's input bits by using a circuit extension, the consistency of the evaluator by modifying the OT extension of [24] and the selective failure attack by a novel combination of the OT extension and the use of the free XOR approach in the garbled circuit. These constructions are made to scale perfectly in parallel.

3 High Level Description

We now give a short, high level, description of our protocol and implementation approach. In broad terms the protocol we present is a merge of the protocols presented in [21] and [24] along with a few novel constructions to efficiently eliminate the security issues the cut-and-choose approach gives, while making all steps as scalable in parallel as possible. In overall terms the protocol can be described as follows:

1. Given a statistical security parameter, ℓ, expressing that the probability of a complete break down is at most $2^{-\ell}$, along with a layer-wise specification of the circuit to be computed, C, the constructor extends the description to get a new circuit, C', that includes a consistency check. Using the description, C',

the constructor then constructs $\ell' = 3.22 \cdot \ell$ (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]) GCs in parallel, one layer at a time. Here, the keys of a given circuit is constructed using a global key and all randomness in the circuits are based on some bytes of randomness. We call this, and the next step, for the *construction phase*.4.

2. The constructor hashes each of the ℓ' circuits' garbled computation tables along with the keys for the evaluators output. She then sends the resultant digests to the evaluator. These digests makes it possible to avoid sending half of the gabled computation tables. This approach is mentioned in [15].
3. The constructor then sends both of the keys of the evaluator's output wires to the evaluator.
4. The constructor and evaluator then engage in OT in order for the evaluator to learn the keys in correspondence with his input for all of the ℓ' circuits. We call this the *OT phase*.
   (a) The creator and evaluator complete a modified OT extension so that the evaluator learns random bitstrings that map to his input bits and the constructor learns the same random bitstrings, along with an equal amount of other random bitstrings that map to the opposite of each of the evaluator's input bits. That is, the OTs will be a 1-out-of-2 OTs of random bitstrings.
   (b) From the circuit generation the constructor will have a 0 and 1 key for each wire. 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.
   (c) The evaluator uses these bitstrings to find the correct input keys for the circuits by a simple XOR operation.
5. The evaluator openly selects $\ell'/2$ circuits for verification and asks the constructor to send the random bits used to generate these particular circuits[3]. We call this and the following three steps for the *cut-and-choose phase*.
6. Using the random bits the evaluator generates the circuits' garbled computation tables and along with the keys or the output wires and verifies that they are correct by hashing them and checking equality to the digests he has already received. That is, he basically just completes the circuit generation phase, but with the seeds of randomness pre-specified, and computes a hash of the results.
7. From the seeds the evaluator also generates the input keys for the GCs. He uses these keys, the differences he received after the OT phase along with his outputs from the OT phase, to reconstructs both the 0 and 1 keys which should be the bitstrings the constructor learned after the OT phase. He hashes each of these and verifies that they are consistent with the digests he got from the constructor at the end of the OT phase. This check verifies that the constructor sent the correct differences after the OT phase.
8. After these checks the constructor sends, to the evaluator, 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.
9. The evaluator degarbles the circuits to achieve the output keys along with their respective external values. He then checks consistency of these outputs. We call this the *evaluation phase*.
10. If the consistency check passes, then the evaluator maps the output keys to their corresponding bits and thus learns his output. If the constructor is also supposed to receive some output, then the evaluator sends the output wires' keys for the constructor's output, to the constructor.
11. The parties then 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.

[3] If one wish to prove the protocol UC secure then these circuits need to be chosen by a coin-flipping protocol, that is, randomly and not by the choice of the evaluator.

The Garbled Circuit. We construct the GCs in very much the same manner as described in [30]. However, we use the optimizations for free XOR [16], garbled row reduction [23] along with an efficient encryption function using permutation bits [27]. Thus our circuit garbling is the same as in [21]. Our optimization arrives in the way we construct and evaluate the GCs in a scalable parallel manner.

First of all, we modify the circuit for 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 follows:

$$f(x, y) = (f_1(x, y),\ f_2(x, y))$$

with $|x| = \tau_a$, $|y| = \tau_b$ and $f_1(x, y)$ being the output the constructor is supposed to learn and $f_2(x, y)$ being the output the evaluator is supposed to learn. Now 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. Or more formally:

$$\mathbf{M}_{i,j} = r[i + j].$$

Using this matrix we now define new function f':

$$f'((x, s),\ (y, r)) = (f_1(x, y),\ (f_2(x, y),\ t))$$

where $s \in_R \{0, 1\}^\ell$, $r \in_R \{0, 1\}^{\tau_a + \ell}$ and $t \in \{0, 1\}^\ell$ and the computation of t is defined as follows:

$$t = (\mathbf{M} \cdot x) \oplus s$$

assuming all binary vectors are in column form.

This means that the new function computes exactly 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. Furthermore, 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. 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.

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 same instruction, multiple data (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, i.e. executing a straight line program.

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, just 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, when having degarbled all gates, and thus found the keys on the output wires, the evaluator verifies consistency as follows:

1. For each of the evaluators outputs, the evaluator uses the output keys previously received by the constructor to find the bits of his output.
2. The evaluator then checks that each of the τ_a digest bits, on all of the $\ell'/2$ circuits, has the same values. That is, the pseudo random verification bits must be equal in all the degabled circuits. If that is not the case then the evaluator outputs failure.

If this test passes then the evaluator takes the majority of the outputs to be his outputs and sends the content of the constructor's output wires to the constructor.

The Modified OT Extension. We use the approach from [24](see Appendix "The OT extension of [24]"), for the base of our modified OT extension. However, we make changes such that as many operations as possible will be hashings of short bitstrings to make it efficiently executable in a parallel machine, and such that we get embedded consistency verification of the evaluator's input choices to eliminate further checks after the OT extension has been executed.

We now define the actual protocol for the modified OT extension. 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 define

$$\psi = 2\tau \cdot \ell',$$

where τ is the amount of bits in the evaluator's input for the modified circuit, i.e. $\tau = \tau_b + \tau_a + \ell$, and ℓ is the statistical security parameter.

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

1. Bob sets the string Γ_B to be the bitstring of his input, y', of size τ, concatenated with itself $2\ell'$ times, that is
$$\Gamma_B = y' \parallel y' \parallel ... \parallel y' = [y']^{2\ell'}.$$
2. Bob then chooses $\left\lceil \frac{8}{3}\kappa \right\rceil$ pairs of seeds, each of κ random bits. That is, for each $i = 1, \ ... \ \left\lceil \frac{8}{3}\kappa \right\rceil$ let $(l_i^0, \ l_i^1) \in_R \{0,1\}^\kappa \times \{0,1\}^\kappa$ be the i'th seed pair.
3. Alice now samples $\left\lceil \frac{8}{3}\kappa \right\rceil$ random bits, $x_1, \ ..., \ x_{\frac{8}{3}\kappa} \in_R \{0,1\}$.
4. Alice and Bob then run $\left\lceil \frac{8}{3}\kappa \right\rceil$ OTs where, for $i = 1, \ ..., \ \left\lceil \frac{8}{3}\kappa \right\rceil$, Bob offers $(l_i^0, \ l_i^1)$ and Alice selects x_i, and receives $l_i^{x_i}$.
5. Now, for each of the $i = 1, \ ..., \ \left\lceil \frac{8}{3}\kappa \right\rceil$ pairs of random bits Bob computes the following two vectors of ψ bits, using $id_{i,j}$ as a unique ID:
$$L_i^0 = \mathrm{H}\left(id_{i,0} \| l_i^0\right) \| \mathrm{H}\left(id_{i,1} \| l_i^0\right) \| \ldots \| \mathrm{H}\left(id_{i,\psi/\kappa} \parallel l_i^0\right),$$
$$L_i^1 = \mathrm{H}\left(id_{i,0} \| l_i^1\right) \| \mathrm{H}\left(id_{i,1} \| l_i^1\right) \| \ldots \| \mathrm{H}\left(id_{i,\psi/\kappa} \parallel l_i^1\right).$$
6. Now, in the same manner Alice extends each of her outputs from the OT from their original length of κ bits, into strings of ψ bits. Thus, Alice computes $L_i^{x_i} = \mathrm{H}\left(id_{i,0} \| l_i^{x_1}\right) \| \mathrm{H}\left(id_{i,1} \| l_i^{x_1}\right) \| \ldots \| \mathrm{H}\left(id_{i,\psi/\kappa} \parallel l_i^{x_1}\right)$.
7. Now, for each $i = 1, \ldots, \left\lceil \frac{8}{3}\kappa \right\rceil$ Bob computes a bitstring, λ_i, and sends these to Alice:
$$\lambda_i = L_i^0 \oplus L_i^1 \oplus \Gamma_B.$$
8. For each $i = 1, \ldots, \left\lceil \frac{8}{3}\kappa \right\rceil$ Alice computes a bit vector as follows
$$L'^{x_i}_i = L_i^{x_i} \oplus (x_i \cdot \lambda_i) = L_i^0 \oplus (x_i \cdot \Gamma_B).$$
9. 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 this to Bob. Furthermore, define $S(\pi) = \{i | i \leq \pi(i)\}$, that is, for each pair, the smallest index is in $S(\pi)$.

10. Now, for all the $\lfloor \frac{4}{3}\kappa \rfloor$ indices $i \in S(\pi)$ do the following:
    (a) Alice computes $d_i = x_i \oplus x_{\pi(i)}$ and sends this to Bob.
    (b) Alice and Bob both compute $Z_i = \left({L'}_i^{x_i} \oplus {L'}_{\pi(i)}^{x_{\pi(i)}} \right)$. 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 Γ_B.
11. 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 the full version of this article), and abort if they are not equal.
12. To check that Bob has constructed Γ_B correctly, i.e. that all the τ bit substrings of Γ_B contains the same bits, Alice and Bob play the following game:
    (a) Alice picks $\lfloor \frac{4}{3}\kappa \rfloor$ random permutations, $\{\phi_i\}_{i=1}^{\lfloor \frac{4}{3}\kappa \rfloor}$, such that $\phi_i : \{1, \ldots, 2 \cdot \ell'\} \to \{1, \ldots, 2 \cdot \ell'\}$ and such that for all j for all i we have $\phi_i(\phi_i(j)) = j$ and $\phi_i(j) \neq j$. Alice sends these to Bob and denote $S(\phi_i) = \{j | j \leq \phi_i(j)\}$, that is, for each pair, for all i we have $S(\phi_i)$ contains the element with the smallest index of the pair.
    (b) Bob views each of the $\lfloor \frac{4}{3}\kappa \rfloor$ strings for $i \in S(\pi)$ as an array of $2\ell'$ blocks, each with τ bits. Call the block with the first τ bits of L_i^0 for $B_{i,1}$, the one with the second τ bits of L_i^0 for $B_{i,2}$ and so on. For each $i \in S(\pi)$, for each index, $j \in S(\phi_i)$, Bob computes $B'_{i,j} = B_{i,j} \oplus B_{i,\ \phi_i(j)}$. For each $i \in \lfloor \frac{4}{3}\kappa \rfloor$ he concatenates these blocks together to one string of $\tau \cdot \ell'$ bits which we call β_i.
    (c) In a similar manner, Alice views each of her $\lfloor \frac{4}{3}\kappa \rfloor$ strings for $i \in S(\pi)$ as an array of ℓ' blocks each with τ bits. Call the block with the first τ bits of string ${L'}_i^{x_i}$ for $A_{i,1}$, the one with the second τ bits of string ${L'}_i^{x_i}$ for $A_{i,2}$ and so on. For each $i \in S(\pi)$, for each index $j \in S(\phi_i)$, Alice computes $A'_{i,j} = A_{i,j} \oplus A_{i,\phi_i(j)}$. She then concatenates these blocks together to one string of $\tau \cdot \ell'$ bits for each i, which we call α_i.
    (d) Finally, Alice and Bob concatenate all their respective strings, $\alpha = \alpha_1 \| \alpha_2 \| \ldots \| \alpha_{\frac{4}{3}\kappa}$ and $\beta = \beta_1 \| \beta_2 \| \ldots \| \beta_{\frac{4}{3}\kappa}$ and evaluate equality on α and β using the protocol for equality of [24], modified for parallel computation (see the full version of this article).
13. For each i and for each $j \in S(\phi_i)$ Alice defines K_j to be the string consisting of the j'th bits from all the strings ${L'}^{x_i}$, i.e. $K_j = {L'}_1^{x_1}[j] \| {L'}_2^{x_2}[j] \| \ldots \| {L'}_{\frac{4}{3}\kappa}^{x_{\frac{4}{3}\kappa}}[j]$. This means that she gets $\tau \cdot \ell'$ keys consisting of $\lfloor \frac{4}{3}\kappa \rfloor$ bits.
14. Now, for each i and for each $j \in S(\phi_i)$ Bob sets M_j to be the string consisting of the j'th bits from all the strings L_i^0, i.e. $M_j = L_1^1[j] \| L_2^0[j] \| \ldots \| L_{\frac{4}{3}\kappa}^0[j]$.
15. Alice lets Γ_A be the string consisting of all the bits x_i for $i \in S(\pi)$, i.e. $\Gamma_A = x_1 \| x_2 \| \ldots \| x_{\frac{4}{3}\kappa}$.
16. Bob now computes $Y'_i = \mathrm{H}(M_i)$ and achieves $(Y'_0, \ldots, Y'_{\tau\ell'})$.
17. Alice computes $X_{i,0} = \mathrm{H}(K_i)$ and $X_{i,1} = \mathrm{H}(K_i \oplus \Gamma_A)$ and achieves $((X_{1,0},\ X_{1,1}), \ldots, (X_{\tau\ell',0},\ X_{\tau\ell',1}))$.

If the parties have been honest it should be the case, that for each $i = 1, \ldots, \tau \cdot \ell'$ $Y'_i = X_{i,\Gamma_B[i]}$.

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

$$\delta_{i,j,0} = X_{i\tau+j,0} \oplus k_{i,j,0},$$
$$\delta_{i,j,1} = X_{i\tau+j,1} \oplus k_{i,j,1},$$

where $k_{i,j,0}$ is the 0-key and $k_{i,j,1}$ 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 $X_{i\tau+j,0}$ or $X_{i\tau+j,1}$, because of the hiding property of the OT. This means that Bob can compute exactly his choice of key, but not the other, because of the security of the free-XOR approach, along with the power of the RO for constructing $X_{i\tau+j,0}$ and $X_{i\tau+j,1}$ so that 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 the following to Bob:

$$\chi_{i,j,0} = \mathrm{H}(X_{i\tau+j,0}),$$
$$\chi_{i,j,1} = \mathrm{H}(X_{i\tau+j,1}).$$

Now, 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

$$\delta'_b = k_b \oplus X_b,$$
$$X'_{\neg b} = \delta_b \oplus \delta_{\neg b} \oplus \Delta,$$
$$\chi'_{\neg b} = \mathrm{H}(X'_{\neg b}).$$

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 has no idea for 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.

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] (described in Appendix "The OT extension of [24]"), are; the non-random construction of Γ_B, the use of hashing to construct the strings $L_i^0,\ L_i^1$ and the addition of Step 12. We now discuss each of these changes one at a time.

Non-random Γ_B. We need Γ_B 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 Γ_B is one-time-padded with the pseudo random strings L_i^0 in Step 7 and 8. It does not compromise the security for Alice either, because of the bit switching in the end of the protocol (Step 13 and 14), along with the fact, that Alice's vector of $x_1, \ldots, x_{\frac{8}{3}\kappa}$ is random, and thus that Bob has no idea if Γ_B will be XORed into an L_i^0 in Step 8.

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 ψ. However, our approach is based on invocations of hash functions on a common seed concatenated with an unique ID. It should here be noted that, even assuming the hash function is a RO, concatenating digests of a common seed (along with a unique ID), as we do, might generally not result in a RO of arbitrary length. Fortunately, it turns out that we, at this step, do not need the powers of a RO, but only a pseudo random generator [24]. Because of this fact the parallel hashing approach remains secure.

Step 12. This step is added to make sure Bob always uses the same value of y' when constructing Γ_B, thus, it is a consistency check. The approach is almost the same as Step 9 to 11. The overall idea is that if Alice is honest then $A'_{i,j} = B'_{i,j}$ for all j. This is so since Γ_B is ℓ' repetitions of the same τ bits. Thus, any two τ bit blocks of Γ_B XORed together will be 0. So, for all i and for all $j \in S(\phi_i)$ we have $x_j \oplus x_{\phi_i(j)} = 0$ and then $A'_{i,j} = B'_{i,j}$ for all $i \in S(\pi)$. However, if Bob tries to cheat, then he can choose to set Γ_B as a string not consisting of repetitions of the bits of y'. If he does so then he has two chances to succeed. First of all, any incorrectly constructed Γ_B will go undetected if he can try to guess when Alice will choose 1, respectively 0, for each $i \in S(\pi)$ in the OTs and then send either a correctly constructed λ_i or an incorrectly constructed λ_i accordingly. Since only half of the OTs are left at this point his success probability is clearly at most $2^{-\lfloor \frac{4}{3}\kappa \rfloor}$.

Besides this attack he can construct Γ_B to have two corrupted segments $y''_1 \neq y'$ and $y''_2 \neq y'$ with $y''_1 = y''_2$ and hope that Alice will choose these segments as one of the pairs in ϕ_i for all i. For each ϕ_i he has $< \ell'^{-1}$ probability of guessing Alice's choice correctly. Thus his probability of guessing correctly for all ϕ_i is $< (\ell'^{-1})^{\lfloor \frac{4}{3}\kappa \rfloor} = 2^{-\lfloor \frac{4}{3}\kappa \rfloor \cdot \log \ell'}$.

Now, by the union bound we get that his total success probability is $\leq 2^{-\lfloor \frac{4}{3} \rfloor \cdot \log \ell'} + 2^{-\lfloor \frac{4}{3}\kappa \rfloor}$. Since κ is the computational security parameter we can assume that $\kappa \geq \ell$ and that $\kappa \geq 2$. From this it clearly holds that $2^{-\lfloor \frac{4}{3}\kappa \rfloor \cdot \log \ell'} + 2^{-\lfloor \frac{4}{3}\kappa \rfloor} < 2^{-\kappa} < 2^{-\ell}$.

Security of the Whole Protocol. As we introduce a new way to combine the OTs and the cut-and-choose GCs, we need to verify that we "fix" the cut-and-choose problems; consistency of the constructor's supplied input keys, consistency of the evaluator's choice of input keys and correctness of the constructor's input to the modified OT extension (selective failure attack). We describe how and why we solve these one at a time. Thus, we do not stringently prove our protocol UC secure, but only sketch why it should be secure in the ROM assuming access to an ideal functionality for OT.

Consistency of the Evaluator's Choices. The consistency of the evaluator's choices actually follows directly from the changes of our modified OT extension and was explained in the previous section.

Consistency of the Constructor's Keys. We remedy the problem that the constructor might send inconsistent input keys for the evaluation circuits, by extending the circuit which we compute with ℓ digest bits as previously explained. This approach is exactly the one described for universal hashing in [22]. That is, the auxiliary input from both parties describe a particular hash function from a family of universal hash functions. Thus this gives us statistical security $2^{-\ell}$ when augmenting the circuit with an ℓ bit digest.

To see that this approach does not leak anything in our usage, assume that the constructor is not consistent with her inputs. In this case two of her inputs, $x \| s \in \{0,1\}^{\tau_a + \ell}$ and $x' \| s' \in \{0,1\}^{\tau_a + \ell}$, will differ in at least one bit. The evaluator can verify that the constructor is consistent by checking that $t =^? t'$. If the two inputs of the constructor differ, then the probability this check will fail is $2^{-\ell}$.

Correctness of the OT Input. For our protocol as it results in random OTs, verifying correctness of the OT input is rather verifying correctness of the differences sent after the OT phase. Now, if the constructor tries to give us the wrong difference strings then the evaluator will detect this with large probability without disclosing his input bits. 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, and thus the overall result of the protocol remains secure.

Parallel Complexity. First see that many of the computationally heavy calculations in the protocol is hashing. Next, notice that these hashes are of "small" bitstrings, bounded by $O(\kappa)$. Now by our approach to parrallizaiton 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 processor the garbling and degarbling time will be boundend by $O(\kappa \cdot d)$ where d is the depth of the circuit to garble.

Regarding the OT extension notice that all the hashes to be computed in a given step of the OT extension can be done independently of each other, and thus completely in parallel.

Now, looking at these steps from each party's point of view, we see that Step 5 is the step requiring most computations for Bob. If he has access to $p \leq \frac{8}{3}\psi \cdot \kappa$ processors then the amount of bits he needs to hash sequentially in the SIMD CREW[4] PRAM model is $O(\psi \cdot \kappa^2/p)$. 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 17. If she has access to $p \leq \psi$ processors then the amount of bits she needs to hash sequentially in the SIMD CREW PRAM model is $O(\psi \cdot \kappa/p)$. 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 $O(\kappa \cdot (\kappa + \psi)) = O(\kappa \cdot (\kappa + \ell \cdot \tau))$ 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 [11] as our hash function and that we use an open GPU optimized implementation of this function due to [7].

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. In fact, we need at least 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 unto 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 in one go, after all the kernels have finished. It should be noted that in order to save memory we only store the 0-key for each wire, since the 1-key can be extremely efficient computed, by simply XORing it with the appropriate global key for a given circuit.

[4] Concurrent read is needed since several processors sometimes need to read the same seed.

Memory Coalescing. We memory coalesced all the data we used, both in the global memory and in the shared memory. As both keys and gabled 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 being the first word of thread 1, the second entry being the first word of thread 2, and so on up to entry 33, which would then contain the second word of thread 1, entry 34 would contain the second word of thread 2 and so on. Thus, all data access is coalesced in a multiple of the warp size.

Restoring from Seed. This is exactly the same as for gate generation except for the fact that the seed used is loaded from a file and not from a random pool.

Evaluation The structure of the kernel is exactly the same as in the case of gate generation, that is, the blocks contain 32 threads, one for each GC it will construct. The grid contains a block for each gate in the current layer of the circuit and there is a kernel launch for each of the layers in the circuit. Before the initial launch the garbled computation table for the whole circuit is copied from the host into global memory along with the initial input keys, one key for each of the 2τ input wires. Finally, a description of the circuit is also loaded into global memory.

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 (from SHA-1) 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 organization 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.

Now, the kernels needed in Steps 5 and 6, and Steps 16 and 17, are a bit different, so we now go through each, one at a time.

Steps 5 and 6. First consider Step 5, this involves hashing $2 \cdot \left\lceil \frac{8}{3}\kappa \right\rceil$ seeds ψ/κ times. First of all, 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 7 in the kernel. To save both memory usage and transfer, 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 ψ bits of the three vectors, L_i^0, L_i^1 and λ_i, and thus contains $\left\lceil \frac{\psi}{32 \cdot \kappa} \right\rceil$ 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 6 proceeds in the same manner, except each block only uses a single seed and each thread only computes a single digest.

Steps 16 and 17. Notice that in these two steps there is not much recycling of data so it does not really matter how we organize the threads. Thus, we simply construct the grid trivially by letting it be 1-dimensional and consists of $\left\lceil \frac{\tau \cdot \ell'}{32} \right\rceil$ blocks, where each block again consists of 32 threads. The job of each thread will then be to compute a digest of a $\frac{4}{3}\kappa$ bit message. In Step 17 each thread will further need to compute an XOR and another digest of $\frac{4}{3}\kappa$ bits.

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 13 and 14. 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^0_i. 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^0_i 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 kb 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 5, 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 the same, commonly used, circuit for oblivious 128 bit AES encryption[5]. This circuit is used as benchmark both in [9,10,21,24], 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 result we ran our experiments with several different statistical security parameters from 2^{-9} to 2^{-159}. 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 while they took place no front end applications were running on either of the machines. These results are summarized in Appendix "Benchmarks" and visualized in figure 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.

[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.

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	(parameter)	Model	Rounds	Time (s)
[10]	Semi honest	-	ROM	$O(1)$	0.20
Our result	Malicious	2^{-9}	ROM	$O(1)$	0.33
Our result	Malicious	2^{-29}	ROM	$O(1)$	0.91
[17]	Malicious	2^{-80}	SM	$O(1)$	1.4
[24]	Malicious	2^{-58}	ROM	$O(d)$	1.6
[17] (Single-core)	Malicious	2^{-80}	SM	$O(1)$	115

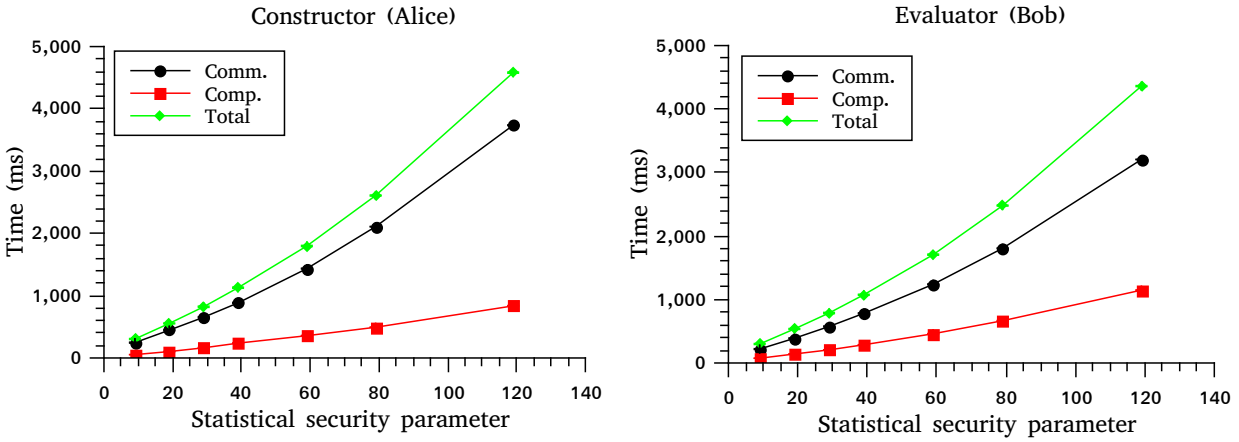

Fig. 1: Timings in milliseconds for both Alice and Bob under different statistical security parameters when computing oblivious 128 bit AES.

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 is clearly not 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].

Next notice that the approach of [24] do beat our results slightly for statistical security of 2^{-58}. However, their round complexity is asymptotically greater than ours which could yield significant performance issues if the protocol where to be executed on the Internet since several packet transmission must be initialized several times during the execution of the protocol. Furthermore, their timings are based on amortization of 54 instances (or 27 is one is happy with 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 a reduction in the statistical

[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.

security. In conclusion, we have showed that the construction of a parallel protocol for 2PC in the SIMD CREW 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 avaible 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.

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. Using cuda for exhaustive password recovery. Unpublished, 2011. `http://http://daimi.au.dk/~jot2re/cuda/index.html`.
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. 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.
10. Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.
11. 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.
12. 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.
13. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In *CRYPTO*, pages 572–591, 2008.
14. 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.
15. Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. 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/`.

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. Lei Xu, Dongdai Lin, and Jing Zou. Ecdlp on gpu. *IACR Cryptology ePrint Archive*, 2011:146, 2011.
30. 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.

Appendix: Benchmarks

Table 2: Timings 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	4.319 ± 0.02345	4.625 ± 0.02585	4.871 ± 0.4558	5.161 ± 0.4755	4.928 ± 0.0473	5.364 ± 0.27725	5.371 ± 0.233981	5.720 ± 0.31789
OT (total)	73.0 ± 7.732	70.1 ± 6.557	115.9 ± 6.656	113.4 ± 6.638	156.2 ± 9.453	144.4 ± 6.910	205.2 ± 7.392	195.7 ± 7.642
OT (comm.)	54.1 ± 7.616	49.2 ± 6.662	85.7 ± 6.389	78.3 ± 6.728	118.2 ± 9.149	102.5 ± 7.126	155.8 ± 7.359	139.9 ± 8.159
OT (comp.)	18.93 ± 0.5959	20.94 ± 0.938	30.22 ± 1.3569	35.15 ± 1.328	38.09 ± 1.6058	41.9 ± 2.215	49.4 ± 0.904	55.9 ± 1.629
Circuits (total.)	227.2 ± 1.088	223.9 ± 3.087	434 ± 2.512	417 ± 2.614	661 ± 3.343	637 ± 6.366	917 ± 3.567	873 ± 5.851
Circuits (comm.)	192.2 ± 1.262	170.3 ± 3.051	361 ± 2.683	310 ± 2.416	536 ± 5.720	473 ± 6.057	729 ± 2.494	642 ± 5.220
Circuits (comp.)	35.0 ± 0.472	53.6 ± 0.671	72.8 ± 0.866	106.4 ± 0.993	125.0 ± 6.480	164.8 ± 1.789	188.7 ± 4.559	231 ± 2.234
Total	304.5 ± 7.921	298.6 ± 8.028	555 ± 7.362	535 ± 7.382	822 ± 10.173	787 ± 9.891	1128 ± 8.842	1074 ± 8.329
(Protocol)	333.6		608.2		912.1		1263	

ℓ	**59**		**89**		**119**	
	Alice	Bob	Alice	Bob	Alice	Bob
IO	6.035 ± 0.45315	6.292 ± 0.31371	6.518 ± 0.1122	7.024 ± 0.5946	7.849 ± 0.3503	8.198 ± 0.52946
OT (total)	273.8 ± 6.774	266.9 ± 6.792	371.2 ± 5.533	363.0 ± 5.620	577.3 ± 5.549	564.3 ± 4.803
OT (comm.)	204.0 ± 7.123	189.2 ± 6.543	277.4 ± 4.693	258.5 ± 5.082	436.5 ± 5.610	407.1 ± 4.304
OT (comp.)	69.82 ± 0.9317	77.79 ± 1.914	93.77 ± 1.6127	104.51± 2.352	140.77± 1.0812	157.2 ± 2.369
Circuits (total.)	1518.7± 5.465	1433.0± 5.060	2231 ± 6.488	2109 ± 4.897	3996 ± 6.031	3789 ± 6.814
Circuits (comm.)	1230.9± 3.622	1050.3± 3.373	1831 ± 4.750	1548 ± 3.314	3301 ± 3.123	2799 ± 7.468
Circuits (comp.)	287.7 ± 5.013	382.7 ± 3.470	399.5 ± 3.338	561.1 ± 3.757	695.6 ± 5.513	990.3 ± 2.494
Total	1798.5± 9.424	1706.2± 8.712	2608 ± 8.543	2479 ± 7.489	4581 ± 9.135	4362 ± 8.269
(Protocol)	1994.0		2878.7		5057.2	

Appendix: 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 an old result due to Damgaard [5].

Appendix: 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 $\frac{8}{3}\kappa$ 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 giving the input and P_1 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:

$$\pi \ : \ \left\{1, \ ..., \ \left\lceil \frac{8}{3}\kappa \right\rceil \right\} \rightarrow \left\{1, \ ..., \ \left\lceil \frac{8}{3}\kappa \right\rceil \right\},$$

such that for all i; $\pi(\pi(i)) = i$. 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 $\lceil \frac{4}{3}\kappa \rceil$ elements which have the smallest index of all pairs in π.
5. Now, for all the $\lceil \frac{4}{3}\kappa \rceil$ 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. $M_j = {Y'}^0_j[1] \parallel {Y'}^0_j[2] \parallel ... \parallel \left({Y'}^0_j\left[\lceil \frac{4}{3}\kappa \rceil\right]\right)$. 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. $\Gamma_A = y_1 \parallel y_2 \parallel ... \parallel y_{\lceil \frac{4}{3}\kappa \rceil}$ 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. $K_j = ({Y'}^0_1 \oplus (y_1 \cdot \Gamma_B))[j] \parallel ({Y'}^0_2 \oplus (y_2 \cdot \Gamma_B))[j] \parallel ... \parallel \left({Y'}^0_{\lceil \frac{4}{3}\kappa \rceil} \oplus \left(y_{\lceil \frac{4}{3}\kappa \rceil} \cdot \Gamma_B\right)\right)[j]$.
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, $((X_{1,0}, \ X_{1,1}) \ ..., (X_{\psi,0}, X_{\psi,1}))$.

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 $Y_i = X_{i,0}$ and if $x_i = 1$ then $Y_i = X_{i,1}$, i.e. a random OT.

Appendix: Multiple Output

Further specification of how to handle output to both parties should have been added. The best approach is probably the one presented in [27]:
Assume the function we wish to compute is called f. The function is augmented such that we instead compute the function f' defined as follows:

$$f'((x, a, b, c), y) = (\lambda, z \parallel f_2(x, y)),$$
$$z = (f_1(x, y) \oplus c) \parallel (a \otimes (f_1(x, y) \oplus c)) \oplus b,$$

where $\otimes$ is the convolution operator modulo two, that is, for bitstrings a and e of length m we have

$$(a \otimes e)[j] = \bigotimes_{i=1}^{m} a[i] \cdot e[i+j].$$

This means that Bob receives both his own output and an encryption of Alice's output along with a MAC on her output. He then sends $(f_1(x,y) \oplus c) \parallel (a \otimes (f_1(x,y) \oplus c)) \oplus b)$ to Alice who uses $a,\ b$ and c to obtain $f_1(x,y)$ and verify that $(a \otimes (f_1(x,y) \oplus c)) \oplus b$ is consistent with $f_1(x,y)$.