Garbled Circuits in the Cloud using FPGA Enabled Nodes

Kai Huang*, Mehmet Gungor*, Xin Fang†, Stratis Ioannidis*, Miriam Leeser*
*Dept of ECE, Northeastern University, Boston, MA
huang.kai1@husky.neu.edu, gungor.m@husky.neu.edu, ioannidis@ece.neu.edu, mel@coe.neu.edu
†Qualcomm, Boxborough, MA
xinfang@qti.qualcomm.com

***Abstract*—Data privacy is an increasing concern in our interconnected world. Garbled circuits is an important approach used for Secure Function Evaluation (SFE); however it suffers from long garbling times. In this paper we present garbled circuits in the cloud using Amazon Web Services, and particularly Amazon F1 FPGA enabled nodes. We implement both garbler and evaluator in software, and show how F1 instances can accelerate the garbling process and rapidly adapt to several different applications. Experimental results, measured on AWS, indicate a 15 times speedup for garbling done using an FPGA. This results in total application speedup, including garbling, communications and evaluation, of close to three times over a large range of application sizes.**

***Index Terms*—privacy, FPGA**

I. Introduction

This research addresses two emerging trends in high performance computing, namely data privacy and Field Programmable Gate Arrays (FPGAs) in the data center.

Data privacy is becoming an increasing concern as our world becomes more and more connected. It is one year since the Europe Union introduced the General Data Protection Regulation (GDPR) and the protection and privacy controls of personal data remain concerns. Many e-commerce businesses have felt the heat to upgrade the way they process, store and analyze personal data. Several US states are following Europe's lead; for example Californias Consumer Protection Act of 2018 goes into effect Jan, 2020. This trend means that cloud users are increasingly aware of and concerned with sharing their data with third parties. This research directly addresses this issue by implementing Garbled Circuits (GC), a technique that allows encrypted data to be processed without being decrypted, and hence providing privacy guarantees regarding user data. However, GC is very computationally expensive and results in significant slow down in processing. To address this, we are using FPGAs in data centers to implement GC.

Increasingly, FPGAs are appearing in HPC systems. Xilinx lists a number of partners in this area including Amazon Web Services (AWS), Baidu, Nimbix and Tencent [1]. Microsoft uses Intel FPGAs in their data centers for Bing searches as well as machine learning applications [2], [3]. The University of Paderborn has several FPGA based research clusters available, and recently announced that it will be acquiring a system from Cray computers with FPGAs [4].

This research accelerates GC using FPGAs in the data center. Specifically, our contributions are:

- An end-to-end implementation of GC on AWS that includes garbler and evaluator implemented on separate nodes.
- An FPGA implementation of the garbler on an AWS F1 instance that shows a 15 times speedup over a large range of sizes of examples.
- End-to-end speedup across a range of size of examples for garbled circuits that show speedup of about 3 times by accelerating garbling on an FPGA.

II. Background

A. Garbled Circuits

Our research accelerates Secure Function Evaluation (SFE), specifically Garbled Circuits (GC), using FPGAs. In this model there are two or more users with data which they wish to keep private, and a function to be evaluated over that data. All parties know the function being evaluated and learn the outcome of the evaluation, but users do not reveal their data. The threat model we follow is "honest but curious" where an adversary follows the protocol as specified, but tries to learn as much as possible. A canonical problem exemplifying SFE is the "Millionaires' Problem:" two millionaires wish to know who is worth more without revealing their personal worth to each other.

Garbled circuits were initially introduced by Yao [5] for two users and has been extended to multiple users. They rely on cryptographic primitives. In the variant we study here (adapted from [6], [7]), Yao's protocol runs between (a) a set of private input owners, (b) an Evaluator, who wishes to evaluate a function over the private inputs, and (c) a third party called the Garbler, that facilities and enables the secure computation.

Garbled Circuits work for any problem that can be expressed as a Boolean circuit. In our and many other implementations, this function is represented as a circuit made up of AND and XOR gates.[1] The Evaluator wishes to evaluate a function f, represented as a Boolean circuit of AND and XOR gates, over

This material is based upon work supported by the National Science Foundation under Grant No. SaTC 1717213.

[1]Recall that AND and XOR gates form a complete basis for Boolean circuits.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

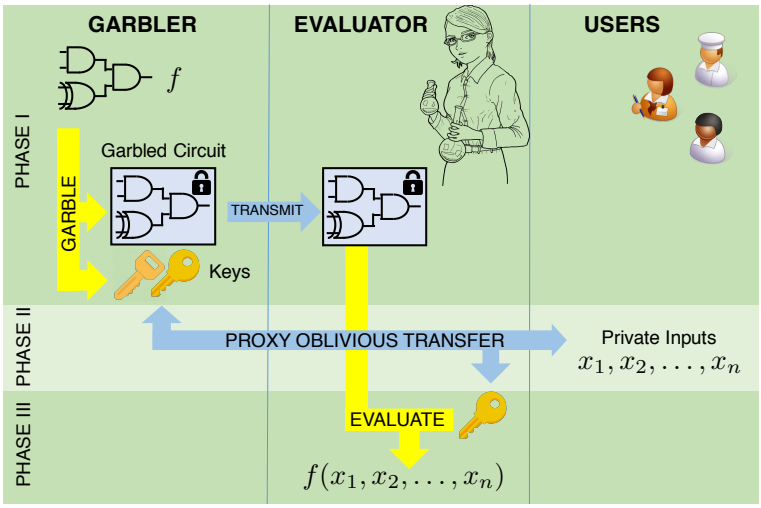
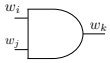


Fig. 1: Yao's Protocol Phases of Operation

private user inputs $x_1, x_2, \ldots, x_n$. We break the problem into three phases, as shown in Fig. 1. In Phase I, the Garbler "garbles" each gate of the circuit, outputting (a) a "garbled circuit," namely, the garbled representation of every gate in the circuit representing f, and (b) a set of keys, each corresponding to a possible value in the string representing the inputs $x_1, \ldots, x_n$. These values are shared with the Evaluator. In Phase II, through proxy oblivious transfer [8], the Evaluator learns the keys corresponding to the true user inputs. In the final phase, the Evaluator uses the keys as input to the garbled circuit to evaluate the circuit, un-garbling the gates. At the conclusion of Phase III, the Evaluator learns $f(x_1, \ldots, x_n)$.

1) Garbling Phase: A function to be evaluated is represented as a Boolean circuit consisting of AND and XOR gates. In the garbling phase, each of these gates is garbled as described in this section. Each gate is associated with three wires: two input wires and one output wire. At the beginning of the garbling phase, the Garbler associates two random strings, $k^0_{w_i}$ and $k^1_{w_i}$, with each wire w_i in the circuit. Intuitively, each $k^b_{w_i}$ is an encoding of the bit-value $b \in \{0,1\}$ that the wire w_i can take.



b_i	b_j	$g(b_i, b_j)$	Garbled value
0	0	0	$Enc_{(k^0_{w_i}, k^0_{w_j}, g)}(k^0_{w_k})$
0	1	0	$Enc_{(k^0_{w_i}, k^1_{w_j}, g)}(k^0_{w_k})$
1	0	0	$Enc_{(k^1_{w_i}, k^0_{w_j}, g)}(k^0_{w_k})$
1	1	1	$Enc_{(k^1_{w_i}, k^1_{w_j}, g)}(k^1_{w_k})$

Fig. 2: A Garbled AND Gate

We describe here how to garble an AND gate. The same principles can be applied to garble an XOR gate, using its respective truth table. We note however that, in practice, XOR gates are handled via the Free XOR optimization [9], discussed in Section II-A3. A garbled AND gate is shown in Fig. 2. For each AND gate g, with input wires (w_i, w_j) and output wire w_k, the Garbler computes the following four ciphertexts, one for each pair of values $b_i, b_j \in \{0,1\}$:

$$Enc_{(k^{b_i}_{w_i}, k^{b_j}_{w_j}, g)}(k^{g(b_i,b_j)}_{w_k}) = SHA(k^{b_i}_{w_i} \| k^{b_j}_{w_j} \| g) \oplus k^{g(b_i,b_j)}_{w_k} \quad (1)$$

Here SHA represents the hash function, $\|$ indicates concatenation, g is an identifier for the gate, and $\oplus$ is the XOR operation. Note that each value k on a wire is implemented with 80 bits in our implementation. The "garbled" gate is then represented by a random permutation of these four ciphertexts. Observe that, given the pair of keys $(k^0_{w_i}, k^1_{w_j})$ it is possible to successfully recover the key $k^1_{w_k}$ by decrypting $c = Enc_{(k^0_{w_i}, k^1_{w_j}, g)}(k^1_{w_k})$ through:[2]

$$Dec_{(k^0_{w_i}, k^1_{w_j}, g)}(c) = SHA(k^{b_i}_{w_i} \| k^{b_j}_{w_j} \| g) \oplus c \quad (2)$$

On the other hand, the other output wire key, namely $K^0_{w_k}$, cannot be recovered. More generally, it is worth noting that the knowledge of (a) the ciphertexts, and (b) keys $(k^{b_i}_{w_i}, k^{b_j}_{w_j})$ for some inputs b_i and b_j yields *only* the value of key $k^{g(b_i,b_j)}_{w_k}$; no other input or output keys of gate g can be recovered. Any Boolean function can be garbled in this manner, by first representing it with ANDs and XORs, and then garbling each such gate.

2) Evaluation Phase: The output of the garbling process is (a) the garbled gates, each comprising a random permutation of the four ciphertexts representing each gate, and (b) the keys $(k^0_{w_i}, k^1_{w_i})$ for every wire w_i in the circuit. At the conclusion of the first phase, the Garbler sends this information for all garbled gates to the Evaluator. It also provides the correspondence between the garbled value and the real bit-value for the circuit-output wires (the outcome of the computation): if w_k is a circuit-output wire, the pairs $(k^0_{w_k}, 0)$ and $(k^1_{w_k}, 1)$ are given

[2]Note that the above encryption scheme is *symmetric* as Enc and Dec are the same function.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

to the Evaluator. To transfer the garbled values of the input wires, the Garbler engages in a proxy oblivious transfer with the Evaluator and the users, so that the Evaluator obliviously obtains the garbled-circuit input value keys $k^b_{w_i}$ corresponding to the actual bit b of input wire w_i.

Having the garbled inputs, the Evaluator can "evaluate" each gate, by decrypting each ciphertext of a gate in the first layer of the circuit by applying equation (2): only one of these decryptions will succeed,[3] revealing the key corresponding to the output of this gate. Each output key revealed can subsequently be used to evaluate any gate that uses it as an input. Using the table mapping these keys to bits, the Evaluator can learn the final output.

3) Optimization: Several improvements over the original Yao's protocol have been proposed, that lead to both computational and communication cost reductions. These include point-and-permute [10], row reduction [11], and Free-XOR [9] optimizations, all of which we implement in our design. Free-XOR in particular significantly reduces the computational cost of garbled XOR gates: XOR gates do not need to be encrypted and decrypted, as the XOR output wire key is computed through an XOR of the corresponding input keys. In addition, the free-XOR optimization fully eliminates communication between the Garbler and the Evaluator for XOR gates: no ciphertexts need to be communicated for these gates. Our implementation takes advantage of all of these optimizations; as a result, the circuit for computing garbled AND gates differs slightly from the garbling algorithm outlined above.

B. AWS

Amazon Web Service (AWS) provides cloud computing, data storage and other data-relevant services for enterprise development, personal use and academic research. Specifically, AWS has f1 type instances that use FPGAs from Xilinx to enable delivery of custom hardware acceleration [12]. We use the f1.2xlarge with Virtex Ultrascale+ ECVU9p FPGAs. The FPGA board includes 4x16 GB external DDR memory.

AWS provides several different ways to program their FPGAs. We use the AWS-FPGA Hardware Development Kit (HDK) which provides development support and runtime libraries for their FPGA instances. The SDK supports OpenCL development; however, since we are developing an overlay architecture it is not a good match for our design.

AWS-FPGA hardware infrastructure connects the FPGA board, which includes external DDR memory, to the host mother board through the PCIe bus. The interconnect with AXI protocol in the hardware design enables data movement between host memory, FPGA on-chip memory and external DDR memory on the FPGA board. Additionally, the software runtime library provides API interfaces to transfer chunks of data to DDR memory and interfaces to access on-chip memory in the FPGA.

[3]This can be detected, e.g., by appending a prefix of zeros to each key $k^b_{w_k}$, and checking if this prefix is present upon decryption.

C. Related Work

Acceleration of garbled circuits on FPGAs is an active area of research. TinyGarble [13] uses techniques from hardware design to implement GCs as sequential circuits and then optimizes these designs. The circuits can be optimized to reduce the non-XOR operations using traditional high-level synthesis tools and simulation. The resulting designs are customized for each problem; thus for each new problem a new bitstream must be generated, hence it is not practical for large designs in a data center setting. We implement as many garbled AND gates as we can keep busy at the same time, and implement garbled circuits directly on top of an efficient overlay, which eliminates the need to recompile the hardware for every new user problem. In MAXelerator [14] the authors implement a very efficient garbling of matrix multiplication in FPGAs. While their design is more efficient for matrix multiplication, ours is more general purpose and supports any problem that a user may wish to garble.

In the previous work, we use an FPGA with a local host for accelerating general garbled circuit problems and demonstrate orders of magnitude improvement over the software version [15]–[17]. The acceleration is achieved via an FPGA overlay architecture, hybrid memory hierarchy, efficient data manipulation and fine grained communication patterns between the host and FPGA. This previous work targets one FPGA with a Stratix V. It only implements the garbler and not the evaluator.

There are several FPGA projects that target AWS f1 instances. In [18], the CAOS framework is extended to integrate with SDAccel running on AWS instances to improve performance. In [19], the authors developed an FPGA-based ultrasonic propagation imaging system to process real-time ultrasonic signals. Firesim [20] builds a cycle-exact simulation platform on large-scale clusters integrated with FPGA accelerators to simulate behaviors of data movement among CPU, caches, DRAM and network switches.

III. Methodology

A complete GC system includes users with inputs, a garbler, an evaluator, and transfer of garbling tables between garbler and evaluator in order for the evaluator to determine the result. Data inputs are communicated from users in encrypted form to the evaluator using oblivious transfer. The implementation described here includes all of these components except for oblivious transfer, which represents a very small portion of the overall run time. In particular, we focus on accelerating the garbler on FPGAs, which is the bottleneck of GC in a data center setting, and show significant speed up.

The garbler garbles the circuit using private input keys which are random strings generated to represent each input. The garbler engages in proxy oblivious transfer with the input owners to send the private keys for the inputs to the evaluator. The evaluator, once it has received the input keys and the garbled tables for the function, can begin to compute the circuit. Note that only the inputs are communicated with OT and this represents a very short part of the communications. In

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

our experiments, keys are generated by the host for verification purposes, so that software and hardware versions generate the exact same strings. The netlist of the circuit is generated using FlexSC [21], which generates a netlist of AND and XOR gates. The XOR gates do not require any cryptographic computation and are therefore considered "free". For AND gates we use SHA-1 cores as described above. The result of garbling is garble tables for each AND gate in the circuit. These tables are communicated to the evaluator, which, together with the input keys, can evaluate the results of the function.

The garbler is much more computationally intensive than the evaluator because it includes more encryption computations. Specifically, for each garbling AND gate, the SHA-1 digest and update functions are computed four times to get the garbling table, which takes the majority of the execution time. Since the SHA-1 core is easily mapped to FPGA hardware and the encryption computations in AND gates can be pipelined and parallelized, the AND and XOR gate operations in the garbler are mapped to FPGA hardware.

Our approach implements a coarse-grained overlay architecture to accelerate GC problems. Garbled AND and XOR gates are implemented on an FPGA along with memory and control for support. Software tools support the mapping of different garbled circuit problems onto this overlay architecture and leverage the interaction between hardware and software while maintaining small communication and memory access overhead. The important aspect of this approach is that we can support large problems and many different problems without reprogramming the FPGA, thus supporting the types of problems to be garbled in a data center setting.

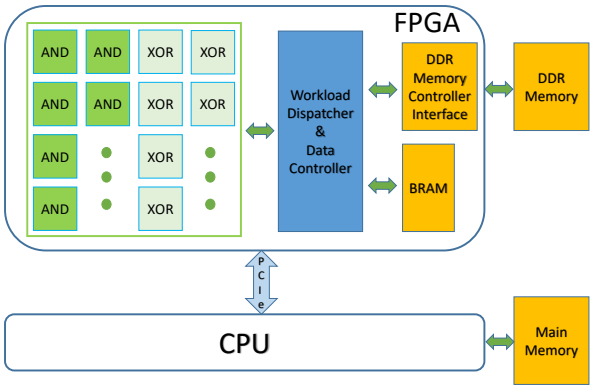


Fig. 3: Hardware Architecture

Fig. 3 shows the hardware architecture for garbled circuit acceleration. The complete design of the overlay architecture includes garbled AND and garbled XOR circuits, a Workload Dispatcher and Data Controller, and a DDR memory interface for accessing the memory where keys are stored. Our design supports a flexible number of AND and XOR gates. We are investigating the optimal number of gates over a wide variety of designs.

The evaluator requires fewer encryption executions and is run as a software program on separate node from the garbler. As soon as the evaluator receives a garbling table from the garbler, it begins to evaluate the circuit. Note that the transfer of garble tables is done in the clear over a high speed network and thus is very fast. When the evaluator is done, it outputs the result of the function being evaluated.

In this research, we have our own Python implementation of garbler and evaluator and we separate these two parts of the system on different nodes and communicate the garble table, resulting in a more realistic scenario. We can also easily compare the garbler implemented on an FPGA to the software implementation. Most importantly, we use FPGAs to implement garbling in parallel and speedup the computation.

IV. Experiments and Results

We conduct our experiments on Amazon AWS. As described above, we run the garbler on F1 instances, while the evaluator is implemented in software. The garbler runs on one AWS node and the evaluator runs on a separate node. Since the computations for the evaluator do not need to be mapped to FPGAs, the evaluator does not require an f1 instance. As described in the previous section, the garbled table is communicated between the garbler and the evaluator using SCP. Note that oblivious transfer (OT) is required for the input values and not the garble tables or circuit layout. In the current implementation we do not implement OT.

For the purpose of validating our designs, we use the same keys for software and hardware implementations. Thus, the initial input keys are generated by the host and transferred to the DDR memory on the FPGA at run time. Intermediate and final output keys are stored in the DDR memory on the FPGA card. These values are not needed after garbling and hence are not communicated beyond the F1 instance.

Our hardware design implements a state machine to handle data movement from DDR memory to FPGA on-chip memory, send the data to the garbled gates on the FPGA and collect the resulting keys and garbling table entries. A GC is composed of GAND and GXOR gates. In our research, we continue to experiment with choosing the optimal number of these gates. Our current design therefore provides a script to generate the state machine and the hardware design for an arbitrary number of garbled AND and XOR gates. In this paper, we set the GAND gate number to 2 and GXOR gate number to 2 in the hardware design. In the future, we will explore the design space and find the optimal one.

We use FlexSC [21] to generate the execution netlist of a circuit, which is a plain text that records all the executed garbled operations consisting of GAND and GXOR gates. We write our own circuit scheduler to distinguish the circuit inputs and determine the layer number for each garbling operation. These operations are performed in a breadth first manner. We then apply a greedy algorithm to schedule the garbled operations (GAND, GXOR) according to the number of gates instantiated on the FPGA and to automatically generate the source C code for netlist mapping and FPGA run time control. The initial memory layout is generated at the same time.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

TABLE I: Timing for total system with python (unit:ms)

applications	Total	garbler	evaluator	gt transfer
16Bit Adder	4.933	3.14	1.793	4.8×10^{-4}
30Bit Ham	18.032	11.686	6.341	4.8×10^{-3}
50Bit Ham	27.811	19.370	8.433	8×10^{-3}
8Bit a*b	30.361	20.473	9.792	9.6×10^{-3}
16Bit a*b	126.366	85.389	40.937	0.0397
32Bit a*b	515.867	349.713	165.993	0.161
64Bit a*b	2066.394	1393.873	671.871	0.650
4Bit Sort10 Number	287.957	182.825	105.065	0.067
4Bit 5x5 Mat Mult	978.663	659.079	319.272	0.312
8Bit 5x5 Mat Mult	4003.290	2684.033	1317.993	1.264
4Bit 10x10 Mat Mult	7984.151	5391.426	2592.123	0.602
8Bit 10x10 Mat Mult	32587.864	22031.146	10546.542	10.176
4Bit 20x20 Mat Mult	65173.249	43868.833	21284.064	20.352

TABLE II: Timing comparing garbler in SW and on FPGA in ms

Application	num. of gates	Garbler(SW)	Garbler(FPGA)	Speed up
16Bit Adder	80	3.140	0.253	12.411
30Bit Ham	330	11.686	0.944	12.379
50Bit Ham	550	19.370	1.550	12.497
8Bit a*b	472	20.473	1.442	14.198
16Bit a*b	1968	85.389	5.840	14.621
32Bit a*b	8032	349.713	23.756	14.721
64Bit a*b	32448	1393.873	95.662	14.571
4Bit Sort10 Number	5531	182.825	15.467	11.820
4Bit 5x5 Mat Mult	15500	659.079	45.479	14.492
8Bit 5x5 Mat Mult	63000	2684.033	184.228	14.569
4Bit 10x10 Mat Mult	126000	5391.426	368.216	14.642
8Bit 10x10 Mat Mult	508000	22031.146	1487.210	14.814
4Bit 20x20 Mat Mult	1016000	43868.833	2966.650	14.787

TABLE III: Timing for total system with software garbler and FPGA garbler in ms

applications	Total(garbler sw)	Total(garbler FPGA)	Speed Up
16Bit Adder	4.933	2.406	2.41
30Bit Ham	18.032	7.290	2.47
50Bit Ham	27.811	9.991	2.78
8Bit a*b	30.361	11.33	2.68
16Bit a*b	126.366	46.817	2.70
32Bit a*b	515.867	189.910	2.72
64Bit a*b	2066.394	768.183	2.69
4Bit Sort10 Number	287.957	120.599	2.39
4Bit 5x5 Mat Mult	978.663	365.063	2.68
8Bit 5x5 Mat Mult	4003.290	1503.485	2.66
4Bit 10x10 Mat Mult	7984.151	2960.941	2.70
8Bit 10x10 Mat Mult	32587.864	12043.928	2.71
4Bit 20x20 Mat Mult	65173.249	24271.066	2.69

We build the hardware implementation for garbled circuit on top of AWS architecture, which provides the basic DMA hardware architecture for data movement among host, FPGA and DDR memory. We report results of timing for different benchmarks built using our versions of the garbler running on an f1.2xlarge instance. We compare the garbling time on an FPGA to the garbling time in Python. The results are shown in Table II. All times are in msecs.

For FPGA garble table generation, we use an overlay architecture consisting of 2 GAND and 2 GXOR gates configured on an AWS f1 large instance. We measure the time from the host CPU including CPU writing to address registers on the FPGA to identify where gates should find their inputs, gates fetching data from DDR memory according to these address registers, processing the data and writing garble table results back to DDR memory. We measure each netlist mapping example 3 times and calculate the average.

Our total run time includes garbling, evaluation and tansfer time over the network. Since it is difficult to measure transfer time on AWS, we estimate it using the network bandwidth of 10 Gbps and assume that we achieve 30% bandwidth.

Consider the system as a whole, where the garbler runs on

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

one node and the evaluator runs on another after it receives the garbling table. The estimate total time is a simple addition of the garbling and evaluation times. The garbling table transfer time is very short compared to the garbling and evaluating time, and does not contribute to the total. In this system, we assume that we see 30% bandwidth to estimate the garbling table transfer delay. If we replace the software garbler by the garbler with FPGA, the garbling time will be shortened while the evaluating time remain as the same. Table III shows the overall time for several examples and the breakdown between garbler and evaluator.

Table II shows the speedup for the Garbler implemented on the FPGA. We are achieving close to 15 times speedup independent of the number of gates in the application. Table III shows total system speedup using the FPGA implementation of the garbler, again with a consistent speedup of around 2.7x. This is end-to-end application speedup that we achieve as circuit examples grow large.

V. Conclusions and Future Work

In this paper, we are mapping the garbling circuit garbler to FPGAs because it is more computationally expensive and the encryptions can be more efficiently executed with parallelism on FPGAs. We can generate our hardware design with an arbitrary number of GAND and GXOR gates on top of AWS's DRAM-DMA architecture and schedule the gate operations to these FPGAs. Our experiments show that we achieve a speed up of close to 15x for the garbler and 2.7x for the whole system. These speedups are consistent over a wide range of application sizes.

We have several enhancements planned for the future. We will add oblivious transfer and all communications times to our experiments, and thus provide more realistic timing for the bottlenecks of the system. We plan to improve on our garbled implementation design by making more efficient use of the memory hierarchy and storing intermediate values in BlockRAM on chip, including intermediate keys that are generated. Finally, our goal is to solve large problems to demonstrate scalability. Thus, we plan to map the garbler to a cluster of hosts integrated with FPGAs and to explore multiple degrees of parallelism in clusters with FPGAs in the data center.

Acknowledgements

This research was supported in part by a grant from Amazon Web Services.

References

[1] "Reconfigurable acceleration in the cloud," 2019. [Online]. Available: https://www.xilinx.com/products/design-tools/cloud-based-acceleration.html

[2] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014, pp. 13–24.

[3] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[4] "Cray to build FPGA-accelerated supercomputer for paderborn university," 2019. [Online]. Available: https://insidehpc.com/2018/04/cray-build-fpga-accelerated-supercomputer-paderborn-university/

[5] A. Yao, "How to generate and exchange secrets," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1986.

[6] M. Naor, B. Pinkas, and R. Sumner, "Privacy preserving auctions and mechanism design," in *1st ACM Conference on Electronic Commerce*, 1999.

[7] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, "Privacy-preserving matrix factorization," in *ACM CCS*, 2013.

[8] M. Naor and B. Pinkas, "Efficient oblivious transfer protocols," in *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2001, pp. 448–457.

[9] V. Kolesnikov and T. Schneider, "Improved Garbled Circuit: Free XOR Gates and Applications," in *ICALP*, 2008.

[10] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. ACM, 1990, pp. 503–513.

[11] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams, "Secure two-party computation is practical," in *ASIACRYPT*, 2009.

[12] Amazon, "Amazon ec2 f1 instances," 2017. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[13] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "Tinygarble: Highly compressed and scalable sequential garbled circuits," in *IEEE S & P*, 2015.

[14] S. U. Hussain, B. D. Rouhani, M. Ghasemzadeh, and F. Koushanfar, "Maxelerator: Fpga accelerator for privacy preserving multiply-accumulate (mac) on cloud servers," in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, p. 33.

[15] X. Fang, S. Ioannidis, and M. Leeser, "Secure function evaluation using an fpga overlay architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 257–266.

[16] X. Fang, "Privacy preserving computations accelerated using fpga overlays," Ph.D. dissertation, Northeastern University, 2017.

[17] X. Fang, S. Ioannidis, and M. Leeser, "Garbled circuits for preserving privacy in the datacenter," in *International Workshop on Heterogeneous High-performance Reconfigurable Computing*, 2016.

[18] L. Di Tucci, M. Rabozzi, L. Stornaiuolo, and M. D. Santambrogio, "The role of cad frameworks in heterogeneous fpga-based cloud systems," in *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 423–426.

[19] S. H. Abbas, J.-R. Lee, and Z. Kim, "Fpga-based design and implementation of data acquisition and real-time processing for laser ultrasound propagation," *International Journal of Aeronautical and Space Sciences*, vol. 17, no. 4, pp. 467–475, 2016.

[20] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra *et al.*, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 29–42.

[21] X. Wang and K. Nayak, "FlexSC," 2014. [Online]. Available: https://github.com/wangxiao1254/FlexSC