FASE: FPGA Acceleration of Secure Function Evaluation

Siam U Hussain
University of California, San Diego
Email: siamumar@ucsd.edu

Farinaz Koushanfar
University of California, San Diego
Email: farinaz@ucsd.edu

Abstract—**We present FASE, an FPGA accelerator for Secure Function Evaluation (SFE) by employing the well-known cryptographic protocol named Yao's Garbled Circuit (GC). SFE allows two parties to jointly compute a function on their private data and learn the output without revealing their inputs to each other. FASE is designed to allow cloud servers to provide secure services to a large number of clients in parallel while preserving the privacy of the data from both sides. Current SFE accelerators either target specific applications, and therefore are not amenable to generic use, or have low throughput due to inefficient management of resources. In this work, we present a pipelined architecture along with an efficient scheduling scheme to ensure optimal usage of the available resources. The scheme is built around a simulator of the hardware design that schedules the workload and assigns the most suitable task to the encryption cores at each cycle. This, coupled with optimal management of the read and write cycles of the embedded memory on FPGA, results in a minimum 2 orders of magnitude improvement in terms of throughput per core for the reported benchmarks compared to the most recent generic GC accelerator. Moreover, our encryption core requires 17% less resource compared to the most recent secure hardware realization of GC.**

I. Introduction

Privacy is one of the most scrutinized topics of this decade. With the increasing popularity of web services, especially those based on machine learning (ML) models [1]–[5], maintaining user data privacy has become a critical issue. Till present, privacy is primarily based on trust, which clearly is not acceptable as evident by several breaches in recent years. We need to develop privacy-preserving frameworks based on provably secure protocols, formally called the Secure Function Evaluation (SFE) protocols [6], [7] that allows two parties to compute a joint function without revealing their respective inputs. Among a number of such protocols in use at present, the most effective one has been Yao's Garbled Circuit [8]. Upon its arrival in 1986, it was mostly considered to be a theoretical concept until its first realization, Fairplay [9], in 2004. Over the last decade a number of enhancements, both in theoretical aspects [10]–[13] and practical realizations [14]–[18], has made real-life employment of the protocol into a reality. However, its usage in large scale machine learning and data mining applications still remains a challenge. As a significant step towards addressing this challenge, we present FASE: FPGA Acceleration of Secure Function Evaluation.

FASE is developed to facilitate the cloud servers to provide secure services to a large number of clients in parallel. In GC, the underlying function is represented as a Boolean circuit, called a *netlist*. The truth-tables of that netlist is encrypted, and the computation is performed on the encrypted netlist. Generation and communication of these encrypted tables between the server and the client cause large overhead compared to the plain-text computation. For a cloud server that is communicating simultaneously to a large number of clients through parallel channels, efficient generation of the encrypted tables becomes a challenge. As we show in this paper, generating them on our FPGA accelerator brings down the protocol execution time within the practical limit.

A number of recent works [19]–[21] have accelerated GC with FPGAs. A secure MIPS processor is presented in GarbledCPU [20], where the netlist is always the Boolean circuit of the processor, upon which the binary of the secure function is loaded. This allows the user the ease of programming in any suitable language. However, it pays the price by having to execute a large netlist. GarbledCPU provides three versions with trade-off between speed and privacy, and even in the least secure version, the overhead is too high for practical purposes.

MAXelerator [19] presented an FPGA accelerator for the GC execution of a Multiply-Accumulate (MAC) for matrix-multiplication, which is the basic building block of a large number of ML models. While it achieves high throughput by custom-designing this specific application, its usage is limited in many scenarios. In one of their own case studies, when applied to the privacy-preserving recommendation system presented in [22], an acceleration of the MAC operation by $\sim 50\times$ results in only $1.5\times$ acceleration of the overall process since only 2/3rd of the operations involved MAC. Our GC accelerator FASE, supporting any generic function, does not achieve such a high improvement on one specific operation (in this case MAC), however, for the same problem, the overall process is accelerated by $\sim 12\times$.

A generic GC accelerator on FPGA is presented in [21]. However, this design was not able to utilize the full capability of the underlying hardware for a couple of reasons. First, it employs very simple scheduling of the Boolean gates that may lead to a large number of encryption units being unused for a significant time throughout the operation. Second, it does not involve any pipeline and therefore incurs a large time gap between consecutive inputs to the encryption units. More importantly, it employs SHA-1 for encryption, which is considered not to be secure anymore [23]–[25]. The authors claim that it is adequate for preserving privacy in the context of garbled circuits, where cryptography is applied at many levels. However, such a statement without a formal security proof is not acceptable and may lead to serious security breaches.

In FASE, we employ AES [26] for encryption similar to all the recent GC realizations on either software or hardware, especially after the appearance of the fixed key block cipher optimization presented by JustGarble [11]. We also optimize the realization of the AES core specifically for GC and achieve around 17% reduction in resource usage per core compared to MAXelerator or GarbledCPU, two of the most recent secure realization of GC. Our pipelined architecture allows the encryption cores to receive one gate each cycle. To ensure the optimal usage of the cores, i.e., minimum idle cycles, we design a scheduling algorithm built around a software simulator for our FPGA accelerator. Moreover, we design a memory management wrapper around the embedded memory to ensure optimal use of the limited read/write ports. As a result, FASE demonstrates minimum 2 orders of magnitude improvement in terms of throughput per core over [21].

In brief, the contributions of this work are,

- We present a pipelined garbling framework that is able to receive one gate every cycle. This allows us to garble multiple gates in parallel using a single garbling core.
- We optimize the encryption core, AES, exclusively for the GC protocol. This results in 17% reduction in resource usage compared to the most recent secure FPGA realization of GC.
- We design an efficient scheduling scheme for our pipelined architecture built around a simulator of the FPGA design. It ensures near optimal use of the encryption cores under the constraints of gate dependency and memory access collision.
- We achieve minimum 2 orders of magnitude improvement in terms of throughput per core compared to the most recent generic GC accelerator on FPGA.

The source code of FASE is available at [27].

II. Preliminaries

A. Oblivious Transfer.

Oblivious Transfer (OT) [28] is a cryptographic protocol between the sender *Alice* and the receiver *Bob*. In a 1-out-of-2 OT protocol, Alice holds 2 messages (x_0, x_1) and Bob holds a selection bit $s \in \{0, 1\}$. Bob receives x_s without revealing s to Alice and learns nothing about x_{1-s}.

B. Garbled Circuit.

Yao's Garbled Circuit (GC) [8] is a cryptographic protocol that allows two parties *Alice* and *Bob* to jointly compute a function $y = f(a, b)$ on their private inputs: a from Alice and b from Bob. Following are the steps of this protocol:

1) The function f is represented as a Boolean circuit, called *netlist*, consisting of 2-input 1-output logic gates.
2) For each wire w in the netlist, Alice assigns two k-bit[1] random keys X_w^0 and X_w^1 associated with the two possible values 0 and 1, respectively.

[1] k is a security parameter, its value is set to 128 in recent works [11], [29]

3) For each gate, Alice generates the *garbled table* by encrypting the keys associated with the values in the output row with respective input keys.
4) Alice then sends the garbled tables along with the keys associated with the values of the bits of her input a.
5) Bob obtains the keys associated with the values of the bits of his input b through 1-out-of-2 OT protocol.
6) Bob uses these keys to decrypt the garbled tables gate by gate and learn the decryption keys of the subsequent gates.
7) Finally, after Bob obtains the keys corresponding to the output y, Alice shares the mapping of keys to the Boolean values, and Bob shares the decrypted keys, and together learn the actual value of y.

Steps (2) and (3) together forms the *Garbling* operation and Step (6) is the *Evaluation* operation.

Sequential Garbling [14]. If the Boolean circuit representing f is sequential, as introduced in TinyGarble [14], steps (2) to (7) is repeated for the number of cycles the circuit needs to complete the computation. Before the start of every cycle (before step (2)) an extra step is added where the input keys of the Flip-Flops (FF) are copied to the output keys.

C. Garbled Circuit Optimizations

Point and Permute [30]. Alice appends a 1-bit random *mask* to each key. The *masked value* of a wire is its actual value XOR with the mask. Alice arranges the rows in the garbled table according to the masked values. Bob learns the masked values of the output wires. In step (7), Alice and Bob communicate only the masked value and mask bit to learn the actual value of the output.

Free XOR [12]. Alice generates a $(k-1)$-bit key R known only to her. For each wire w, she generates the key X_w^0 and sets $X_w^1 = X_w^0 \oplus (R \parallel 1)$($\parallel$ denotes concatenation). With this convention, the key for the output wire r of an XOR gates with input wires p, q can be simply computed as $X_r = X_p \oplus X_q$. Thus XOR, XNOR, and NOT gates do not need to be garbled.

Row Reduction [13]. Instead of generating the key for the output wire of a gate randomly, it is computed as a function of the keys of the inputs such that the first entry of the garbled table are all 0s and no longer needs to be sent. Thus it reduces the number of rows in the garbled table from four to three.

Half Gate [10]. This optimization breaks a non-XOR gate into two half-gates for which one party knows one input. It employs both free XOR and row reduction such that each half-gate can be garbled with single encryption. As a result, the size of the non-XOR gate truth table is reduced to two rows.

Garbling with a Fixed-key Block Cipher [11]. This method allows to efficiently garble non-XOR gates using fixed-key AES with a unique identifier for each gate. In this scheme, the output key X_r is encrypted with the input keys X_p and X_q according to following the function

$$E(X_r, X_p, T, X_q) = \pi(K) \oplus K \oplus X_r, \qquad (1)$$

where, $K = 2X_r \oplus 4X_p \oplus T$, T is a unique gate identifier, and π is a fixed-key block cipher (instantiated with AES).

III. Global Flow

A. Security Model and Terminology

In accordance with most of the recent realization of the GC protocol [14]–[18] we adopt the *honest-but-curious* security model, which assumes that both parties follow the protocol honestly yet may try to learn additional information from the information at hand. We use the term XOR gates to refer to XOR, XNOR and NOT gates, and the term non-XOR gates to refer to all other gates (e.g., AND, OR, NAND, etc). In addition to these gates, our GC framework supports D Flip-Flops (DFFs) with inputs I, and D and output Q. At reset, The value at input I passes to Q, otherwise, at each positive edge of the clock, value at input D passes to Q. The Boolean circuit representing the function f being executed through GC is referred to as the *netlist*, and the circuit that we design on FPGA to generate the garbled tables is referred to as the *circuit*. The term *netlist cycle* is used to refer to the clock cycles pertaining to the netlist, and the term *cycle* is used to refer to the clock cycles pertaining to the circuit on FPGA.

B. Client-Server Model

FASE accelerates the generation of garbled tables on a cloud server and is transparent to the clients acting as the evaluators. Garbling and evaluation are similar tasks, and our garbling engine can also act as the evaluator engine with few tweaks. However, as outlined in [19], acceleration of the computation is only beneficial to cloud servers since they are communicating with a large number of clients in parallel. For a client, communicating with only one server, the speedup of the overall GC execution does not justify having a hardware accelerator. The presence of FASE on the server side is invisible to the clients except for the speed up in service.

C. Netlist Format

The netlist is the Boolean representation of the function $y = f(a, b)$, where a, and b are inputs from the server and the client respectively. The netlist file holds information of the number of netlist input bits (i.e., the total number of bits in a and b), the numbers of FFs, XOR and non-XOR gates in the netlist, and the indices of the gates generating the final output y. It also holds information of the input and output indices, and the Boolean logic of each gate. In addition, it may also have *stall* entries indicating that the inputs of the next gate are not ready in the current cycle.

JustGarble [11] introduced the SCD format to represent the netlist. The SCD format employs efficient indexing of the gates and wires that results in a compact file. However, it requires access to multiple elements of the arrays at the same time which is not amenable to the embedded memory used to store the netlist on FPGA. FASE uses the indexing format of the SCD file but stores the netlist in a new HSCD format shown in Table I that supports reading the netlist in streaming style. D, I, INPUT_0, and INPUT_1 are indices of the inputs to the DFFs and gates respectively. LOGIC holds the 4 output bits of the gate's truth table where the inputs are in the order 00, 01, 10, 11. IS_OUTPUT is a one-bit value that is set to 1 if the DFF or gate's output is connected to the netlist output y. The index of the gate's output wire is the index of the gate in this list, thus does not need to be stored explicitly.

TABLE I
HSCD FORMAT TO STORE THE NETLIST

# of Lines	Content
4	Netlist parameters (input and output bit lengths, number of dffs, gates etc)
# of dffs	D || I || 1111 || IS_OUTPUT
# of gates	INPUT_0 || INPUT_1 || LOGIC || IS_OUTPUT
# of stalls	- || - || 0000 || 0

D. Execution Steps of FASE

Our implementation is distributed over two platforms: the host CPU and FASE on the FPGA together act as the garbler. The netlist is generated at the host CPU and transferred to FASE. This step is performed only once per function, irrespective of the number of clients or the number of executions. Then for each client, the following steps are performed.

1) FASE generates R (free-XOR, Sect II-C) and the AES key (fixed-key block cipher, Sect II-C) and sends to the host.
2) FASE generates keys for constant values 0 and 1 and sends them to the host. These keys are used if the initial values of the DFFs are assigned to constants.
3) For each netlist cycle
   a) For each DFF
      i) If this is the first netlist cycle, the keys for the Q input of the FFs are assigned either to a constant key (corresponding to 0 or 1 depending on the value) or to the key of input I. In the latter case, the input keys are generated and sent to the host.
      ii) For the rest of the netlist cycles, keys for the Q inputs of the DFFs are copied from keys at the D inputs.
   b) For each gate
      i) If the inputs of the gate are connected to the netlist inputs a or b, FASE generates the keys corresponding to those inputs and sends them to the host.
      ii) FASE generates the garbled table and output key and sends the garbled table to the host.
      iii) If the output of the gate is connected to the netlist output y, the mask bit (Point and Permute, Sect II-C) is stored to an internal register file.
   c) At the end of each netlist cycle, all the mask bits are transferred to the host.
4) The host CPU performs the communication with the client, including OT, and jointly compute the output y.

Note that, generation of the garbled tables is independent of the inputs a or b. Therefore, FASE does not need any information from the host after the netlist is transferred. On the other hand, the host CPU receives the garbled tables for each non-XOR gate and the mask of each bit of the output y. However, the key of the output of each gate is only used internally inside FPGA to generate the garbled tables and outputs of the subsequent gates and not sent to the host.

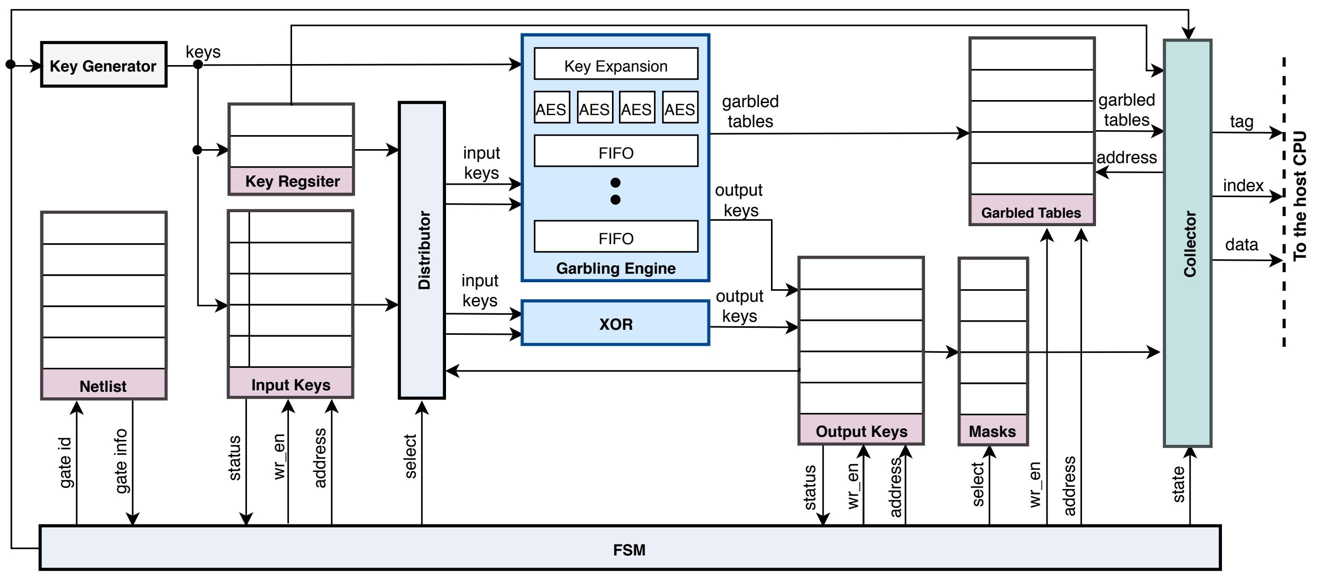

Fig. 1. Architecture of FASE

IV. Architecture of FASE

Fig 1 shows the different components of FASE. The heart of the system is the pipelined garbling engine that is capable of receiving one gate per cycle. Its inputs and outputs are stored in six different memories: Netlist, Key Register, Input Keys, Output Keys, Masks, and Garbled Tables. Three of them are dedicated to storing the keys. The Key Register stores the two most recently generated keys. The Input Keys memory stores the keys associated with the netlist inputs a and b. The rest of the keys, generated either by the garbling engine or XOR is stored in the memory named Output Keys. Efficient synchronous management of these memories is key to the optimal usage of the encryption cores inside the garbling engine. The garbling operation is executed by the control logic, consisting of the Finite State Machine (FSM) and the distributor, according to the steps described in Sect III-D. The collector works in parallel to the control logic to collect and transfer the generated data from FASE to the host CPU. In addition, FASE incorporates a key generator which is the source of the random keys associated with the netlist inputs.

A. *Key Generator*

The key generator consists of $2K$ True Random Number Generators (TRNG), each of which generates 1 random bit per cycle. The TRNGs are implemented with sets of ring oscillators following the design presented in [31]. The clocks to the TRNGs are controlled through two clock buffers each connected to K TRNGs. Each set of TRNGs is only enabled when new keys need to be generated as described in Sect III-D.

B. *Garbling Engine*

Given the two keys associated with the value 0 of the two inputs and the Boolean logic of the gate, the garbling engine generates the garbled table and the key associated with the value 0 of the output. Note that according to the free-XOR optimization, the key for the value 1 of a wire is generated by XORing the key for value 0 with R. The garbling engine incorporates both the row-reduction and half-gate optimizations and thus the generated tables have two rows per gate. No garbled tables are generated for the XOR gates. Output keys of these gates are generated by the XOR block.

The garbling engine has four AES cores. They have a 10 stage pipelined architecture. Therefore, generating the garbled table and output key of each non-XOR gate requires 10 cycles. However, the garbling engine can accept one gate per cycle due to the pipelined architecture. Even though it increases the throughput by a large margin compared to the previous accelerators, it also creates a dependency issue since the output keys of the gate, g_n sent to the garbling engine at the n-th cycle is not ready until $n+10$-th cycle. Therefore, from cycles $n+1$ to $n+9$ only the gates that are independent of the output of g_n can be sent to the garbling engine. We solve this issue by smart scheduling elaborated in Sect V.

As shown in Eq 1, the encrypted key is again XORed with the input keys and the gate identifier. These keys are passed through 10-stage FIFOs to be XORed with the AES output.

The AES encryption function accepts two inputs: the plaintext, which is a function of the input keys, and the AES key. According to the fixed key block cipher optimization [11], the AES key is fixed for all the AES cores for one garbling session. This allows us to instantiate only one common key expansion module for all the four AES cores. As a result, the resource utilization by the garbling engine is reduced by around 17%. In our implementation, the key expansion module has 5 pipeline stages. To ensure that the expanded key is ready before the garbling of the first gate starts, we insert idle states if necessary (i.e., if the number of DFFs in the netlist is less than 4) between step (1) and (3-b) of the steps described in Sect III-D.

C. Control Logic

The control logic consists of the FSM and the distributor. The distributor controls the source of the input keys fed to the garbling engine. The FSM performs the following tasks:

- Fetch the current gate from the netlist and determine the source of its input keys.
- If the source is the Input Keys memory, and not already generated, turn on the key generator.
- If the source is the Output Keys memory, and not already computed, stall the operation.
- If both the input keys of the current gate is ready, increment the gate index to fetch the next gate.
- Determine the source of the output keys for each gate based on the gate logic (`XOR` or non-`XOR`).
- Control the storing of the masks based on the IS_OUTPUT value for the gate.
- Once all the garbled tables are computed, reset the gate index and increment the netlist cycle.

D. Memory Management

Netlist. This is implemented on a BRAM that stores the netlist in the HSCD format presented in Sect III-C. BRAMs have one cycle latency from receiving the input address (gate index) to providing the data (input indices and logic of the gate). Before proceeding to the next gate, i.e. increasing the gate index, the control logic needs to read the input indices of the current gate for checking if the input keys are ready. This would result in a latency of 2 cycles per gate. Note that the previous GC accelerators did not need to deal with this issue. MAXelerator [19] performed only one specific function and the netlist of that function was embedded into its control logic. The generic accelerator in [21] arranged the circuits in layers of independent gates and only garbled gates of one layer at one time. This approach results in FPGA resources being left unused for a large part of the operation.

Fortunately, with the indexing format of SCD [11], the gates are accessed sequentially. Therefore, the gate index is always incremented by 1. We design a wrapper around the BRAM, that always reads one address ahead of the given address (gate index) and stores the data into a register. Whenever the address is incremented, the data already stored in the register is provided at the output and the next data is requested from the BRAM. From the perspective of the control logic, this is equivalent to reading from a register file or a distributed RAM, that provides the read data immediately.

Key Register and Input Keys. The Input Keys is a dual-port BRAM to store the keys associated with the netlist inputs a and b. If the input of the current gate is connected to either of these, a new K bit key is generated and stored in the memory, if not already generated. Otherwise, the key is read from the memory. To keep track of whether or not the key is already generated, a register file of 1-bit *flag* registers with the same depth as the BRAM is maintained. To avoid the possibility of collisions through the read and write ports of the BRAM, the most recent pair of keys are stored in the Key Register, a register file with two K-bit registers. There is a write to the Input Keys memory only if a new key is generated, and in that case, the keys to the garbling engine are supplied from the Key Register, eliminating the possibility of collision.

Output Keys. The keys associated with gate outputs generated by the garbling engine or `XOR` are stored in a dual port BRAM. These keys are also read later for subsequent gates that depend on the current gate. Unlike the Input Keys, flag registers are not required for the Output Keys since the readiness of the required keys at a certain cycle is pre-computed offline for each netlist, and encoded in the HSCD file. In [21], four cycles are required per gate for BRAM access. Reducing this time to two cycles is straight forward- using a dual port, instead of single port BRAM. However, for each gate, two keys are read from the memory and one key is written back. Therefore, theoretically, it is possible to process each gate in 1.5 cycles. To reduce the total memory access time we design a wrapper, as shown in Fig 2, around the BRAM.

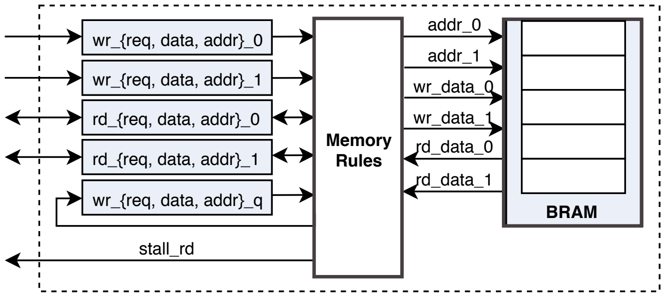

Fig. 2. Wrapper module around the BRAM of Output Keys.

In addition to the external read and write request, address and data for the two ports 0 and 1, it has an additional output *stall_rd* that directs the control logic to stall the operation. Moreover, it has an internal queue that can hold one write command. The read and write ports of the BRAM are controlled according to the following rules.

- If there is no queued request,
  - if the number of write requests is more than or equal to the number of read requests, the write requests are performed and the read requests are stalled.
  - if the number of write requests is less than the number of read requests, the read requests are performed and the write request is queued.
- If there is a write command in the queue,
  - the read commands are stalled irrespective of the number of read requests.
  - if the number of write requests is 2, the write command at port 1 is queued and the queued write command is performed through that port.
  - if the number of write requests is 1, the queued write command is performed through the free port.

A write request is never queued for more than one cycle. Queuing of a write command is invisible to the control logic.

In addition to these, the garbled tables are stored in a dual port BRAM. The 1-bit output masks are stored in a register file so that all the mask bits can be transferred to the host CPU in one or two cycles.

E. Collector

The collector performs the communication with the host CPU. Four types of data are sent from FASE to the host: (i) the R and AES keys, (ii) keys for the netlist inputs, (iii) garbled tables for the non-XOR gates, and (iv) output masks for the netlist outputs. At each cycle, the collector sends the following three pieces of information to the host:

1) A *tag* indicating the type of the data being sent.
2) The *index* of the respective data.
3) The *data*.

The keys for the netlist inputs are assigned the highest priority since only the most recent pair of keys are stored in the Key Registers. The garbled tables are stored in a dual port BRAM. One of the port is used to write the garbled tables. The collector uses the other port to read them. Since the gates are accessed sequentially, the garbled tables are also written sequentially. Therefore, the read address being smaller than the write address indicates that there are new garbled tables that need to be transferred. After all the garbled tables are sent, all the output masks are sent together in one or two cycles depending on the bit length of the netlist output y.

Communication Bandwidth. Let us define the following netlist parameters. The total input bit-width is M, the output bit-width is N, the total number of gates is G, the total number of XOR gates is X, and the netlist takes C cycles to compute the operation Then the data transferred from the FPGA to the host CPU to garble one netlist is $4K+((M+2(G-X))K+N)C$ bits. This is significantly less compared to the accelerator presented in [21], which needs to transfer $(3G+3(G-X))K$ bits per netlist, the difference being $(3G+(G-X)-M)K-N$. Note that [21] only supports combinational circuit, therefore the number of netlist cycles C is always 1, but the number of gates and the number of input bits will be larger. Eventually, the product $C \times G$ will be of the same order for both combinational and sequential netlists of the same function. Moreover, [21] does not support half-gate optimization, therefore the garbled table has three rows instead of two.

V. Scheduling the Gates

As explained in Sect IV-B, the garbling engine is able to accept one new gate per cycle. However, since each gate takes 10 cycles to process, the gates sent to the garbling engine at cycles $n+1$ to $n+9$ should be independent of the gate g_n sent at cycle n. If not properly scheduled, there may be a large number of idle cycles, when the control logic waits for the input keys of the current gate being computed. We treat this problem as offline scheduling of a Directed Acyclic Graph (DAG) to a Bounded Number of Processors (BNP) with the number of processors set to the number of pipelined stages [32]–[36]. The scheduling is performed in two steps:

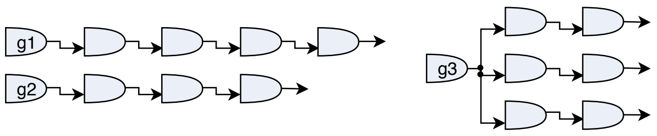

Fig. 3. Different types of gate dependencies.

1) The gates are ordered according to their priority.
2) From the ordered list, the gates are assigned one by one to one of the free processors.

A. Setting the priority

In the majority of the work on DAG scheduling, one of the three parameters are used as the measure of priority:

- t-level: length of the longest path (excluding the gate) from the netlist input to the gate.
- b-level: length of the longest path (including the gate) from the gate to netlist output.
- ALAP: length of the critical path - b-level.

The key difference between assigning tasks to parallel processors and to a single pipelined processor is that in the latter case the bottleneck is not the availability of the processors rather the readiness of the inputs. Therefore, the last two parameters are better than the t-level in this case since they both prioritize the gates with a higher number of dependent gates. However, they still do not result in the optimal ordering of gates. This is illustrated by the small example netlist in Fig 3. According to both b-level and ALAP, gates $g1$ and $g2$ have higher priority than $g3$, while scheduling $g3$ first will free up more number of gates. In this work, we employ the *weighted fanout* of a gate as a measure of its priority. The fanout of a gate is the number of gates dependent on it. In computing weighted fanout the weight of XOR gates are set to 1, and the weights of the non-XOR gates are set to 10, the number of cycles it takes to compute their output keys.

B. Adding Gates to the Queue

To add gates to the queue from the ordered list we follow Algorithm 1. This is a simulator of the hardware architecture presented in Sect IV. It includes all the constraints of the hardware (e.g., the number of pipeline stages, processing one gate per cycle, memory conflict, etc.) except one. That is at every cycle, instead of reading only one gate, it reads all the gates that have not been queued yet from the ordered list and queue the first ready gate. A gate is ready when the keys assigned to its inputs have been computed. If none of the gates are ready, it inserts a $stall$.

The input to the algorithm is the ordered list of gates $GateList$ of size G. Every element of $GateList$ is a gate $g(i_0, i_1, l)$, where $i_0, i_1 \in WireList$ are the indices of the two input wires and l is the Boolean logic of g. $WireList$ is the list of wires that are ordered according to the following rules (introduced in SCD format [11]): (i) the first M indices belong to the M input wires of the netlist, and (ii) the index of the output wire of a gate is the sum of the gate's index

Algorithm 1: Algorithm to assign gates to the queue

```
input      : Ordered list of gates $GateList$
output     : The queue of gates $Q$
parameters: number of input wires $M$
            number of gates $G$
            number of pipeline stages $P$
1  create arrays $W_0$, $W_1$, $R_0$, $R_1$ of 0s
2  create an array $Ready$ of 0s
3  for $k = 1$ to $M$ do
4      set $Ready[k]$ to $P$
5  end
6  $c$ = 0
7  while size of $Q < G$ do
8      increment $c$
9      for $k = 1$ to $M + G$ do
10         if $Ready[k]$ is not 0 then
11             increment $Ready[k]$
12         end
13     end
14     $stall\_rd$ =
15     $mem\_rules(W_0[c-1], W_1[c-1], R_0[c-1], R_1[c-1])$
16     if $stall\_rd$ is true then
17         push $stall$ into $Q$
18         continue
19     end
20     for $k = 1$ to $G$ do
21         read $g(i_0, i_1, l)$ from $GateList[k]$
22         if $Ready[i_0] > P$ and $Ready[i_1] > P$ then
23             push $g$ into $Q$
24             set $R_0[c]$ to 1, set $R_1[c]$ to 1
25             if $l$ is XOR then
26                 set $W_0[c+1]$ to 1
27                 set $Ready[k]$ to $P$
28             else
29                 set $W_1[c+P+1]$ to 1
30                 set $Ready[k]$ to 1
31             end
32             break
33         end
34     end
35     push $stall$ into $Q$
36 end
```

in the $GateList$ and M. The task mem_rules at line 15 of Algorithm 1 decides if there is a stall in the read operation according to the rules outlined in Sect IV-D.

Note that scheduling instructions in a pipelined processor is an active area of research [37]–[39]. However, these schemes target real-time scheduling. Therefore, they primarily optimize the speed of scheduling and deal within only a limited view of the different sets of operations running in parallel. In the case of GC, the gates are scheduled offline, only once per netlist, and the scheduler has the complete view of the entire netlist. Therefore, these schemes do not benefit this specific task.

VI. Evaluation

A. Benchmark Functions

Table II shows the benchmark functions along with the number of gates and XOR gates and the number of netlist cycles to complete each function used to evaluate FASE. These benchmarks, except the MACs, are the largest sequential netlists provided in the TinyGarble [14] repository, one of the most recent and efficient netlist synthesis tools for GC. The netlists for multiplication performs the same functions as those in [14], but we use different implementations that favor parallelism. The MAC netlists perform the same function as the custom GC accelerator of MAXelerator [19].

TABLE II
Benchmark Functions

Benchmark	Function	Input bits	# Netlist csycles	# Gates	# XORs
Mill_8_8	Millionaire's	8	8	4	3
Add_8_1	Addition	8	1	37	30
Add_8_8	Addition	8	8	5	2
Hamm_32_1	Hamming dist.	32	1	188	157
Hamm_32_32	Hamming dist.	32	32	13	8
Hamm_512_512	Hamming dist.	512	512	21	12
Mult_256_512	Multiplication	256	512	1699	1186
Mult_1024_2048	Multiplication	1024	2048	6782	4735
MAC_8_1	MAC	8	1	397	231
MAC_16_1	MAC	16	1	1678	1077
MAC_32_1	MAC	32	1	7036	4805
CORDIC_32_31	Trigonometric	32	31	2464	1544
AES_128_11	AES	128	11	4662	3225

B. Resource Utilization

FASE is implemented on a Xilinx Virtex UltraScale VCU108 (XCVU095) FPGA. The resource utilization on this platform is shown in Table III. In this implementation, the number of gates $G = 2^{13}$, the number of input bits $M = 2^{10}$, and the number of output bits $N = 2^8$. These parameters are selected such that FASE supports the largest of the benchmark functions presented in Table II. The memory requirement will increase with the increase in the values of G, M, or N. However, the resource utilization by the garbling engine and the key generator is independent of these parameters. Therefore, we report the independent utilization by the latter components separately in Table III. The maximum supported clock frequency on this platform is 200MHz.

TABLE III
Resource Utilization of FASE

Resource	Total		Garbling Engine		Key generator	
	Num	%	Num	%	Num	%
LUT	50035	9.31	31330	5.83	18202	3.39
FF	11416	1.06	5612	0.52	3917	0.36
LUTRAM	569	0.74	553	0.72	0	0.00
BRAM	68.5	3.96	0	0.00	0	0.00

We do not compare the resource utilization with previous GC accelerators. MAXelerator [19] supports only one specific function. The accelerator in [21] employs SHA1 for encryption, which is not considered secure anymore, and 80-bit keys instead 128 bits used by the recent GC realizations. Therefore, it is not possible to make a fair comparison.

C. Evaluation of Scheduling and Memory Management

The goal of these optimizations is to reduce the number of cycles per gate. According to our evaluation, using weighed fanout instead of ALAP results in 0 to 9% reduction in the percentage of idle cycles. We choose to compare to ALAP as it has been shown to be superior over other methods of offline scheduling in a bounded number of processors [35].

To evaluate the performance improvement provided by the memory management techniques presented in Sect IV-D, we compare the average number of cycles per gate for different benchmark functions over one netlist cycle without and with the memory management in Table IV. The table shows that memory management reduces the average number of cycles per gate by up to 0.5. To put these values into context, the theoretical minimum value of cycles per gate is 1.5.

TABLE IV
EVALUATION OF THE EFFECT OF THE MEMORY OPTIMIZATION

Benchmark	Without optimization		With optimization		
	# Cycles	Cycle/gate	# Cycles	Cycle/gate	Improv.
Mill_8_8	19	4.75	17	4.25	0.50
Add_8_1	120	3.24	120	3.24	0.00
Add_8_8	17	3.40	17	3.40	0.00
Hamm_32_1	342	1.82	320	1.70	0.12
Hamm_32_32	65	5.00	65	5.00	0.00
Hamm_512_512	113	5.38	113	5.38	0.00
Mult_256_512	3123	1.84	2686	1.58	0.26
Mult_1024_2048	12492	1.84	10744	1.58	0.26
MAC_8_1	769	1.94	675	1.70	0.24
MAC_16_1	3160	1.88	2770	1.65	0.23
MAC_32_1	12746	1.81	11414	1.62	0.19
CORDIC_32_31	4554	1.85	4047	1.64	0.21
AES_128_11	9004	1.93	7697	1.65	0.28

D. Comparison with Previous Work

We now compare the performance of FASE with the two most recent GC accelerators implemented on FPGA [19], [21]. Both these works employ multiple cores while FASE employs a single core with a pipelined architecture. Similar to [19], we compare the performances on a per core basis. Table V compares the throughput of these works with FASE for the reported benchmark functions. Throughput is computed as the number of garbled netlist per core per cycle. FASE shows 110× to 310× improvement in throughput compared to the generic accelerator [21]. Table V also shows that the throughput of FASE is 3.65× to 4.98× smaller compared to [19] for MAC operation. We would like to emphasize that [19] is a customized architecture that can only perform this one specific function while FASE is a generic GC accelerator capable of executing any given netlist.

E. Improvement in Throughput over Software Approach

Finally, we evaluate the performance of FASE against the software realization of GC presented in TinyGarble [20]. TinyGarble is built on the JustGarble [11] framework. With the fixed key block cipher optimization, this is the fastest software realization of GC at present. We run it on an Intel Xeon E5-2600 processor @ 2.9GHz with 128 GB memory. Since there is a large difference in the clock frequency of the FPGA and the CPU, we compare the performance in terms of absolute time in us, instead of the number of cycles. As shown in Table VI, FASE is up to 19× faster than the fastest software realization of GC.

TABLE V
COMPARISON OF FASE WITH PREVIOUS GC ACCELERATORS

Benchmark	Previous Work	# cores	# cycles	# cycles of FASE	Improv.†
Millionaire (2)	[21]	43	1.90E+02	6.70E+01	121.94
Addition (6)	[21]	43	5.60E+02	9.90E+01	243.23
Hamming (10)	[21]	43	1.20E+03	1.66E+02	310.84
Hamming (30)	[21]	43	2.20E+03	3.38E+02	279.88
Hamming (50)	[21]	43	2.80E+03	6.69E+02	179.97
A * B (8)	[21]	43	4.40E+03	6.19E+02	305.65
A * B (32)	[21]	43	3.60E+04	1.05E+04	147.78
A * B (64)	[21]	43	1.10E+05	4.28E+04	110.51
MAC_8_1	[19]	8	2.40E+01	7.01E+02	1/3.65
MAC_16_1	[19]	14	4.80E+01	2.80E+03	1/4.17
MAC_32_1	[19]	24	9.60E+01	1.15E+04	1/4.98

†In terms of (number of netlists garbled per cycle per core)

As mentioned earlier, the customized GC accelerator for MAC presented in [19] accelerates the privacy-preserving recommendation system in [22] by only 1.5× even though it accelerates the MAC operation, which is 2/3rd of all the computations, by ∼50×. In that particular work, most of the remaining operations involved trigonometric functions which can be executed by the CORDIC function. FASE accelerates MAC by ∼13× and CORDIC by ∼10×. Therefore, it is able to accelerate the system in [22] by ∼12×.

TABLE VI
COMPARISON OF FASE ON FPGA WITH TINYGARBLE [14] ON CPU

Benchmark	Garbling Time(cc)	Garbling Time (μs)		
	FASE	TG [14]	FASE	Improv.
Mill_8_8	2.59E+02	1.04E+01	1.30E+00	8.05
Add_8_1	1.75E+02	8.92E+00	8.75E-01	10.19
Add_8_8	1.38E+02	1.34E+01	6.90E-01	19.37
Hamm_32_1	3.38E+02	2.98E+01	1.69E+00	17.64
Hamm_32_32	2.72E+03	1.09E+02	1.36E+01	8.00
Hamm_512_512	7.01E+04	1.98E+03	3.51E+02	5.65
Mult_256_512	1.64E+06	8.27E+04	8.19E+03	10.10
Mult_1024_2048	5.61E+07	1.25E+06	2.81E+05	4.46
MAC_8_1	7.01E+02	3.82E+01	3.51E+00	10.90
MAC_16_1	2.80E+03	1.63E+02	1.40E+01	11.62
MAC_32_1	1.15E+04	7.38E+02	5.73E+01	12.87
CORDIC_32_31	1.29E+05	6.82E+03	6.44E+02	10.60
AES_128_11	8.77E+04	5.07E+03	4.38E+02	11.57

VII. CONCLUSION

We have presented FASE, an FPGA accelerator for Secure Function Evaluation (SFE) by employing the Yao's GC protocol. FASE employs a pipelined garbling engine, efficient assignment of gates to the engine to reduce idle cycles, and optimized memory management to increase the number of gates garbled per cycle. As a result, it achieves at least 2 orders of magnitude improvement in throughput per core compared to the most recent generic GC accelerator on FPGA. FASE also outperforms the customized GC accelerators when applied to problems requiring diverse computations.

Acknowledgments. This work was supported in parts by the NSF(CNS-1619261), SRC(2016-TS-2690), MURI(FA9550-14-1-0351), and ONR (N00014-17-1-2500) grants.

REFERENCES

[1] F. Sebastiani, "Machine learning in automated text categorization," *ACM computing surveys (CSUR)*, vol. 34, no. 1, pp. 1–47, 2002.

[2] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[3] H. Aradhye, W. Hua, and R.-S. Lin, "Native machine learning service for user adaptation on a mobile platform," Apr. 23 2013, uS Patent 8,429,103.

[4] J. Kirk, "Ibm join forces to build a brain-like computer," http://www.pcworld.com/article/2051501/, 2016.

[5] N. Jones *et al.*, "The learning machines," *Nature*, vol. 505, no. 7482, 2014.

[6] Y. Huang, D. Evans, and J. Katz, "Private set intersection: Are garbled circuits better than custom protocols?" in *NDSS*, 2012.

[7] Brenner, Perl, and Smith, "hcrypt SFE project," https://hcrypt.com/sfe/.

[8] A. Yao, "How to generate and exchange secrets," in *Foundations of Computer Science, 1986., 27th Annual Symposium on*, 1986.

[9] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay – a secure two-party computation system," in *USENIX Security*. ACM, 2004.

[10] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole: Reducing data transfer in garbled circuits using half gates," Cryptology ePrint Archive, 2014, http://eprint.iacr.org/2014/756.

[11] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013.

[12] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2008.

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

[14] 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.

[15] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, "Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation," in *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 2016, pp. 112–127.

[16] B. Kreuter, A. Shelat, B. Mood, and K. Butler, "PCF: A portable circuit format for scalable two-party secure computation." in *USENIX Security*, 2013.

[17] A. Rastogi, M. A. Hammer, and M. Hicks, "WYSTERIA: A programming language for generic, mixed-mode multiparty computations," in *S&P*. IEEE, 2014.

[18] N. Büscher, M. Franz, A. Holzer, H. Veith, and S. Katzenbeisser, "On compiling boolean circuits optimized for secure multi-party computation," *Formal Methods in System Design*, vol. 51, no. 2, pp. 308–331, 2017.

[19] S. U. Hussain, B. D. Rouhani, M. Ghasemzadeh, and F. Koushanfar, "MAXelerator: FPGA Accelerator for Privacy Preserving Multiply-Accumulate (MAC) on Cloud Servers," in *DAC*. ACM, 2018.

[20] E. M. Songhori, S. Zeitouni, G. Dessouky, T. Schneider, A.-R. Sadeghi, and F. Koushanfar, "Garbledcpu: a mips processor for secure computation in hardware," in *DAC*. ACM, 2016.

[21] X. Fang, S. Ioannidis, and M. Leeser, "Secure function evaluation using an fpga overlay architecture." in *FPGA*, 2017.

[22] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, "Privacy-preserving matrix factorization," in *Conference on Computer & communications security*. ACM, 2013.

[23] S. Manuel, "Classification and generation of disturbance vectors for collision attacks against sha-1," *Designs, Codes and Cryptography*, vol. 59, no. 1-3, pp. 247–263, 2011.

[24] X. Wang, Y. L. Yin, and H. Yu, "Finding collisions in the full sha-1," in *Annual international cryptology conference*. Springer, 2005, pp. 17–36.

[25] A. Satoh, "Hardware architecture and cost estimates for breaking sha-1," in *International Conference on Information Security*. Springer, 2005, pp. 259–273.

[26] N. F. Pub, "197: Advanced encryption standard (aes)," *Federal information processing standards publication*, vol. 197, no. 441, 2001.

[27] S. U. Hussain. FASE: FPGA Acceleration of Secure Function Evaluation. [Online]. Available: https://github.com/siamumar/FASE

[28] M. Naor and B. Pinkas, "Computationally secure oblivious transfer," *Journal of Cryptology*, vol. 18, no. 1, 2005.

[29] X. Wang, S. Ranellucci, and J. Katz, "Authenticated garbling and efficient maliciously secure two-party computation," in *CCS*. ACM, 2017.

[30] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *Symposium on Theory of computing*. ACM, 1990.

[31] K. Wold and C. H. Tan, "Analysis and enhancement of random number generator in fpga based on oscillator rings," *International Journal of Reconfigurable Computing*, vol. 2009, 2009.

[32] M. L. Pinedo, *Scheduling: theory, algorithms, and systems*. Springer, 2016.

[33] K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *2010 31st IEEE Real-Time Systems Symposium*. IEEE, 2010, pp. 259–268.

[34] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

[35] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, no. 3, pp. 381–422, 1999.

[36] W. Bożejko, *A new class of parallel scheduling algorithms*. Oficyna wydawn. Politechniki Wrosłwskiej, 2010.

[37] J. R. Goodman and W.-C. Hsu, "Code scheduling and register allocation in large basic blocks," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, 2014, pp. 88–98.

[38] J. P. Shen and M. H. Lipasti, *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[39] A. Benoit, Ü. V. Çatalyürek, Y. Robert, and E. Saule, "A survey of pipelined workflow scheduling: Models and algorithms," *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, p. 50, 2013.