Secure Function Evaluation Using an FPGA Overlay Architecture

Xin Fang
Dept of Electrical and Computer Engineering
Northeastern University
Boston, MA, USA
fang.xi@husky.neu.edu

Stratis Ioannidis
Dept of Electrical and Computer Engineering
Northeastern University
Boston, MA, USA
ioannidis@ece.neu.edu

Miriam Leeser
Dept of Electrical and Computer Engineering
Northeastern University
Boston, MA, USA
mel@coe.neu.edu

ABSTRACT

Secure Function Evaluation (SFE) has received considerable attention recently due to the massive collection and mining of personal data over the Internet, but large computational costs still render it impractical. In this paper, we leverage hardware acceleration to tackle the scalability and efficiency challenges inherent in SFE. To that end, we propose a generic, reconfigurable implementation of SFE as a coarse-grained FPGA overlay architecture. Contrary to tailored approaches that are tied to the execution of a specific SFE structure, and require full reprogramming of an FPGA with each new execution, our design allows repurposing an FPGA to evaluate different SFE tasks without the need for reprogramming. Our implementation shows orders of magnitude improvement over a software package for evaluating garbled circuits, and demonstrates that the circuit being evaluated can change with almost no overhead.

Keywords

FPGA; Secure Function Evaluation; Garbled Circuits

1. INTRODUCTION

Mining behavioral data is a ubiquitous practice among Internet companies, and is presently happening at an unprecedented scale. Google, Netflix, Amazon, and Facebook routinely monitor and mine a broad array of behavioral signals collected from their users, including ad clicks, pages visited, and products purchased. Such information is monetized through targeted advertising or personalized product recommendations. Behavioral data collection is therefore of considerable business value to online companies [2]; moreover, there are often benefits to society at large: mining such data can aid in the detection of medical emergencies or the spread of diseases [33], in polling to assess political opinions [1] or news adoption [21], in the assessment of terrorist threats [35], etc. On the other hand, this massive data collection and mining has given rise to significant privacy concerns, extensively documented by researchers [20, 24, 26, 30, 34, 36, 40] as well as the popular press [2, 42]. Such concerns are only likely to further increase with the emergence of the "Internet of things", as wearable devices and home automation sensors connected to the Internet proliferate.

This state of affairs gives rise to the following challenge: given the benefits of mining behavioral data to both online companies and the society at large, is it possible to *enable data mining practices without jeopardizing user privacy*? A series of recent research efforts [4, 6, 11, 27–29] have attempted to address this issue through cryptographic means and, in particular, through secure function evaluation (SFE). SFE allows an interested party to evaluate any desirable polynomial-time function over private data, while revealing only the answer and nothing else about the data. This offers a strong privacy guarantee: an entity executing a secure data-mining algorithm over user data learns only the final outcome of the computation, while the data is never revealed to the entity. SFE can thus enable, e.g., a data analyst, a medical professional, or a statistician, to conduct a study of sensitive data, without jeopardizing the privacy of the participants (online users, patients, etc.).

Any algorithm to be executed over amounts of data at the scale encountered in the above settings needs to be highly efficient and scalable. SFE over private data therefore poses a significant challenge, as it comes at a considerable additional computational cost compared to execution in the clear. Prior work has made positive steps in this direction, showing that a variety of important data mining algorithms [27–29] can be computed using Yao's Garbled Circuits (GCs) [43, 44] in a parallel fashion. The function to be evaluated is converted to a binary circuit which is "garbled" in such a way that an evaluator of the circuit learns only the values of its output gates. Execution of this circuit is subsequently parallelized, e.g., over threads [28] or across a cluster of machines [27].

Nevertheless, this approach to parallelization leaves much to be desired: for example, in [27], even under parallelization over 128 cores, executing a typical data-mining algorithm like Matrix Factorization through SFE is of the order of 10^5 slower compared to (parallel) execution in the clear. In practice, this means that applying MF to a dataset of 1M entries requires roughly 11 days under SFE, a time largely prohibitive for practical purposes.

In this paper, we advocate leveraging hardware acceleration to tackle the scalability and efficiency challenges in-

FPGA '17 February 22-24, 2017, Monterey, CA, USA

ACM ISBN 978-1-4503-4354-1/17/02.

DOI: http://dx.doi.org/10.1145/3020078.3021746

herent in SFE. FPGAs are by design an excellent hardware platform for the implementation of SFE primitives and, in particular, garbled circuits. This is precisely because FPGAs are tailored to executing nearly identical operations in parallel. The types of operations encountered in garbled circuits (namely, garbling and un-garbling gates) fit this pattern precisely: they involve, e.g., a series of symmetric key encryptions, XORs, and other well-defined primitive operations (see also Section 3). In that sense, an FPGA implementation of SFE benefits from both high speed evaluation and hardware-level parallelization.

On the other hand, the amount of computation required to evaluate a garbled circuit for an application at the usual data-mining scale cannot fit in a single FPGA. For this reason, evaluating a function securely entails partitioning computations into sub-tasks to be programmed and evaluated over a single FPGA. A practical implementation therefore needs to allow repurposing an FPGA to quickly compute different SFEs or different sub-tasks of a larger SFE. For this reason, tailored approaches that are tied to the execution of a specific SFE structure, and require full reprogramming of an FPGA with each new execution, cannot be applied efficiently to the types of SFE problems we wish to address.

To address these challenges, we propose a *generic, reconfigurable implementation of SFE as a coarse-grained FPGA overlay architecture*. As FPGAs have become more dense and capable of holding a large number of gate equivalents, there has been an increased interest in FPGA overlay architectures [5, 13, 14, 16–18, 41]. An FPGA overlay consists of two parts: (1) a circuit design implemented on the FPGA fabric using the usual design flow, and (2) a user circuit mapped onto that overlay circuit. Garbled circuits are excellent candidates for an FPGA overlay design. Precisely because components of a circuit follow a generic structure, an overlay approach that does not reprogram FPGAs from scratch, but simply *reroutes* connections between elementary components (in our case, garbled AND and XOR gates) leads to important efficiency improvements.

This paper makes the following contributions:

- We design and implement a generic FPGA overlay architecture for the execution of arbitrary garbled circuit topologies. In our design, FPGAs are programmed once to contain implementations of garbled components (AND, XOR gates). Wiring and instantiation is determined at execution time through writing to registers and memory. Thus, the overhead for repurposing the FPGA for different circuit computations is kept very low.
- We integrate our implementation with ObliVM [22], a framework mapping code written in a high-level language to a garbled circuit, allowing arbitrary programs written in ObliVM to be mapped to our FPGA overlay architectures.
- We evaluate the performance of our GC overlay architecture on several examples and demonstrate orders of magnitude speedup over ObliVM. We demonstrate the effects of using the overlay architecture, which results in change time for different circuit computations that have little effect on overall performance.

The remainder of the paper is organized as follows. We present related work in Section 2 and background on garbled circuits in Section 3. Our implemented system and overlay architecture are presented in Section 4 and experimental results in Section 5. Finally, we present our conclusions and future work in Section 6.

2. RELATED WORK

Garbled Circuits. Although garbled circuits were proposed by Andrew Yao nearly three decades ago [43, 44], it is only in the last few years that the research community has made progress at improving their efficiency, bringing their application closer to practicality. Several improvements over the original protocol have been proposed, including the point-and-permute [3], row reduction [25], and the Free-XOR [19] optimizations; we use all of these in our implementation.

Building on these optimizations, there has been a surge of recent programming frameworks, such as TASTY [9], FastGC [10], Fairplay [23], and ObliVM [22], that provide software implementations of GCs. These frameworks, particularly ObliVM, allow developers without any cryptography expertise to convert algorithms expressed in a high-level language to GCs. None of these frameworks focus on hardware acceleration. We provide an interface to ObliVM in our work; this allows us to describe algorithms in a high level language, map them to circuits through ObliVM, and then use our software to execute these circuits over our FPGA overlay architecture.

FPGA overlays. Recently, as FPGAs have become more dense and capable of holding a large number of gate equivalents, there has been an increased interest in FPGA overlay architectures. An FPGA overlay consists of two parts: (1) a circuit design implemented on the FPGA fabric using the usual design flow, and (2) a user circuit mapped onto that overlay circuit. Overlays are in general used for two purposes. The first is to create FPGA designs that are independent of the specific structures on a particular FPGA and therefore to make designs portable, or, in other words, able to be mapped to FPGAs from different vendors and to different devices from the same vendor. This class of FPGA overlay designs [5, 41] creates basic FPGA structures, such as Look-Up Tables (LUTs) and routing, built on top of those provided in silicon on the target FPGA chip. We are using an FPGA overlay for the second purpose; namely, to reduce the amount of time to translate a design to an FPGA implementation. FPGAs offer a great deal of reconfigurability and flexibility; however, this comes at the cost of programming. Generating designs that run efficiently on FPGAs can be challenging for the end user. In addition, the *compilation* process for a high end FPGA design can take several hours. Here *compilation* refers to the complete set of steps from specification (in a hardware description language (HDL) or high level language) to generating a bit stream to download to the FPGA. These steps include synthesis, place-and-route and bit stream generation for the target FPGA. Examples of this style of FPGA overlay architecture include Network on a Chip (NoC) overlays [16, 17] and instruction set extension overlays [18]. In these cases, structures are built on the FPGA and the overlay architecture provides flexible routing among them.

Hardware Implementations of Garbled Circuits. Recent studies have used FPGAs as well as GPUs [12, 31] for hardware implementations of garbled circuits. TinyGarble [37] uses techniques from hardware design to implement

GCs as sequential circuits and then optimize these designs, reporting on results using high level synthesis tools and simulation, but do not report any actual hardware implementations. TinyGarble produces more efficient solutions for a single GC instance, but does not handle multiple pieces of a GC or different garbled circuits the way our overlays do. Järvinen et al. [15] target embedded system and describe the first FPGA implementation of GC. This implementation is at a completely different design point than ours: while generic and able to support a wide range of hardware implementations, the proposed FPGA design implements only one encryption core. In contrast, our overlay architecture implements hundreds of encryption cores on a single FPGA and executes them in parallel.

A recent FPGA implementation of garbled circuits by Songhori et al. [38] take a different approach. Rather than implement garbled circuits directly, they implement a *garbled* MIPS core. Problems to be evaluated securely are written in code, that is compiled to MIPS assembler and then run securely on their garbled MIPS processor. The goal of [38] is to fabricate this MIPS core; FPGAs are used for prototyping the design. Using MIPS assembly code to represent the problem being evaluated alleviates the problem of lengthy FPGA place and route cycles. However, the FPGA is not used as efficiently as in our implementation: Songhori et al. use considerably fewer encryption cores, and running code on a MIPs processor creates an extra level of overhead.

Our architecture uses much more parallelism than other FPGA implementations of garbling. For starters, we implement four SHA-1 cores in hardware for each AND gate, while others use one encryption core serially [15]. In addition, 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.

3. TECHNICAL BACKGROUND

Yao's protocol (a.k.a. *garbled circuits*) [43] is a generic cryptographic protocol for secure function evaluation. In short, it allows the secure evaluation of an arbitrary function over private data, provided this function can be represented as a circuit. We give a brief overview of the protocol below (see, e.g., [8], for a detailed treatment).

3.1 Garbled Circuits Overview

In the variant we study here (adapted from [25, 29]), Yao's protocol runs between (a) a set of private input owners (e.g., Google's users), (b) an Evaluator, (e.g., a data analyst working for Google), that wishes to evaluate a function over the private inputs, and (c) a third party called the Garbler, that facilities and enables the secure computation. Formally, let n be the number of input owners, and let $x_i \in \{0,1\}^*$ denote the private input of individual i, $1 \leq i \leq n$, represented as a binary string. Finally, let $f : (\{0,1\}^*)^n \to \{0,1\}^*$ be the function that the Evaluator wishes to compute over the private data. The protocol satisfies the following property: at the conclusion of the protocol, the Evaluator learns *only* the value $f(x_1, x_2, \ldots, x_n)$ and nothing else about $x_1, \ldots, x_n$, while the Garbler learns nothing.

A critical assumption behind Yao's protocol is that the function f can be expressed as a *Boolean circuit*, and, more specifically, as a directed acyclic graph (DAG) of AND and XOR gates.[1] The structure of the circuit – and, thus, the function to be computed – is known to all participants: e.g. the circuit could be computing the sum or the maximum among all inputs x_i.

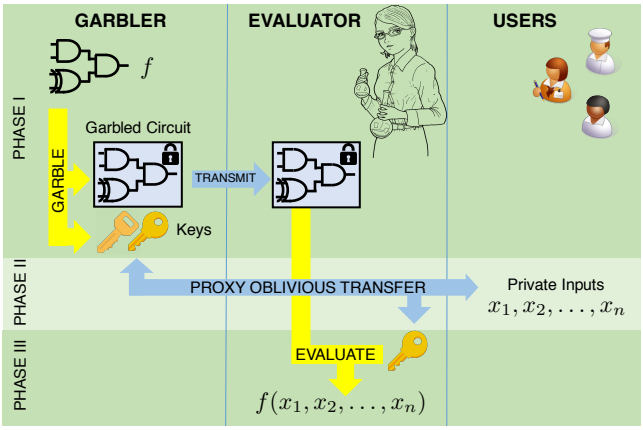


Figure 1: Yao's protocol. The Evaluator wishes to evaluate a function f, represented as a binary circuit of AND and XOR gates, over private user inputs $x_1, x_2, \ldots, x_n$. In Phase I, the Garbler "garbles" each gate of the circuit, outputting (a) a "garbled circuit", namely, the garbled representation of every gate in the circuit representing f, and (b) a set of keys, each corresponding to a possible value of the inputs $x_1, \ldots, x_n$. In Phase II, through proxy oblivious transfer, the Evaluator learns the keys corresponding to the true user input values, while the Garbler learns nothing. In the final phase, the Evaluator uses the keys as input to the garbled circuit to evaluate the circuit, ungarbling the gates in breadth-first order. At the conclusion of Phase III, the Evaluator learns $f(x_1, \ldots, x_n)$.

Overall, Yao's protocol consists of three phases:

1. **Garbling Phase.** During the garbling phase, the Garbler prepares (a) a set of encrypted (i.e., "garbled") truth tables for each binary gate in the circuit, as well as (b) a set of random strings, termed *keys*, one for each possible binary value in the strings representing the inputs. At the conclusion of this phase, the Garbler sends to the Evaluator the garbled truth tables; each such table is referred to as a "garbled gate", and all gates together constitute the "garbled circuit".

2. **Oblivious Transfer Phase.** Subsequently, the Evaluator, Garbler, and the input owners engage in a proxy oblivious transfer [7, 25, 32]. Through this, the Evaluator retrieves the input keys from the Garbler that correspond to true input binary values held by the owners. Oblivious transfer ensures that, although the Evaluator learns the correct keys, the cleartext input values are never revealed to either the Garbler or the Evaluator.

3. **Evaluation Phase.** Finally, the Evaluator uses these input keys to "evaluate" the gates of the circuit, effectively decrypting the garbled gates. Each such decryption reveals a new key that allows the Evaluator to ungarble/decrypt subsequent gates connected to it. Ungarbling the output gates reveals the value $f(x_1, \ldots, x_n)$.

[1] Recall that any Boolean circuit can be represented using only ANDs and XORs.

The above three phases are illustrated in Figure 1. The execution flow (as well as the opportunity for parallelism) is determined by the Boolean circuit representing function f. Both the "garbling" of the gates, that occurs at the Garbler, and the "ungarbling/evaluation", that occurs at the Evaluator, are computationally intensive tasks; these are precisely the operations that we propose to accelerate using FPGAs. We describe these phases in more detail below.

3.2 Garbling Phase

We now describe how gates are garbled in Yao's protocol. As illustrated in Figure 2, each binary gate in the DAG representing the circuit 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. For each gate g, with input wires (w_i, w_j) and output wire w_k, let $g(b_i, b_j) \in \{0,1\}$ be the binary output of the gate given inputs $b_i, b_j \in \{0,1\}$ at wires w_i and w_j, respectively. For each gate g, the Garbler computes the following four ciphertexts, one for each pair of values $b_i, b_j \in \{0,1\}$:

$$\texttt{Enc}_{(k^{b_i}_{w_i}, k^{b_j}_{w_j}, g)}(k^{g(b_i,b_j)}_{w_k}) = \texttt{SHA1}(k^{b_i}_{w_i} \| k^{b_j}_{w_j} \| g) \oplus k^{g(b_i,b_j)}_{w_k}, \quad (1)$$

where SHA1 is the SHA1 hash function, $\|$ indicates concatenation, g is an identifier for the gate, and $\oplus$ is the XOR operation.

The "garbled" gate is then represented by a *random permutation* of these four ciphertexts. An example of a garbled AND gate is illustrated on Fig. 2. 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 $c = \texttt{Enc}_{(k^0_{w_i}, k^1_{w_j}, g)}(k^1_{w_k})$ through[2]:

$$\texttt{Dec}_{(k^0_{w_i}, k^1_{w_j}, g)}(c) = \texttt{SHA1}(k^{b_i}_{w_i} \| k^{b_j}_{w_j} \| g) \oplus c. \quad (2)$$

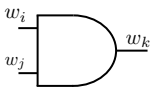
On the other hand, the other output wire key, namely $k^0_{w_k}$, cannot be recovered having access only to $(k^0_{w_i}, k^1_{w_j})$. More generally, it is worth noting that the knowledge of (a) the ciphertexts, and (b) keys $(k^{b_i}_{w_i}, k^{b_j}_{w_j})$ for some inputs b_i and b_j yields *only* the value of key $k^{g(b_i,b_j)}_{w_k}$; no other input or output keys of gate g can be recovered.

At the conclusion of the garbling phase, the Garbler sends the garbled gates (each comprising a random permutation of four ciphertexts) 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 to the Evaluator. Finally, the Garbler keeps the random wire keys $(k^0_{w_i}, k^1_{w_i})$ that correspond to circuit-input wires w_i, i.e., wires at the very first layer of the circuit, encoding user inputs; all other wire keys are discarded.

3.3 Oblivious Transfer Phase

To transfer the correct keys of the circuit-input wires to the Evaluator, the Garbler engages in a proxy oblivious transfer (OT) with the Evaluator and the users. Through proxy OT, the Evaluator obliviously obtains the circuit-input value keys $k^b_{w_i}$ corresponding to the actual bit b of circuit-input wire w_i. Proxy OT ensures that (a) the Garbler does not learn the user inputs, (b) the Evaluator can compute the function on these inputs alone, and (c) the Garbler learns nothing. Note that this OT only pertains to circuit-inputs (the first layer of the circuit); as such, its communication cost is typically several orders of magnitude smaller than the Garbler to Evaluator transfer occurring at the end of the previous phase. As proxy OT is not as intensive in computation or communication as the other two phases, and does involve hardware acceleration, we do not describe it in detail. We refer the interested reader to [7, 25, 32] for a formal description and implementation.

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



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

Figure 2: A garbled AND gate. Each of the three wires w_i, w_j, w_k is associated with two random keys k^0, k^1, representing the value 0 and 1, respectively. The garbled AND gate consists of the four ciphertexts appearing on the rightmost column of the table above: each possible output value key is encrypted using the corresponding input pair of keys, along with a string g representing the gate id. A random permutation of these four ciphertexts is revealed by the Garbler to the Evaluator.

3.4 Evaluation Phase

Having the keys corresponding to true user inputs, the Evaluator can "evaluate" each gate, by decrypting each ciphertext of a gate in the first layer of the circuit through Eq. (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 ungarble/evaluate any gate that uses it as an input. The evaluator can thus proceed ungarbling gates in breadth first order over the DAG blueprint of the Boolean circuit, until finally obtaining the keys of gates at the last layer of the circuit. Using the table mapping these keys to bits, the Evaluator learns the final output.

3.5 Optimizations

Several improvements over the original Yao protocol have been proposed recently, that lead to both computational and communication cost reductions. These include the point-and-permute [3], row reduction [25], and Free-XOR [19] optimizations, all of which we implement in our design. Point-and-permute reduces the four decryptions at the evaluator to one. Row-reduction reduces the number of ciphertexts that need to be transmitted by the Garbler to the Evaluator from four to three. Free-XOR 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

[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. In practice, the point-and-permute optimization of [3] eliminates the need for attempting to decrypt all four ciphertexts.

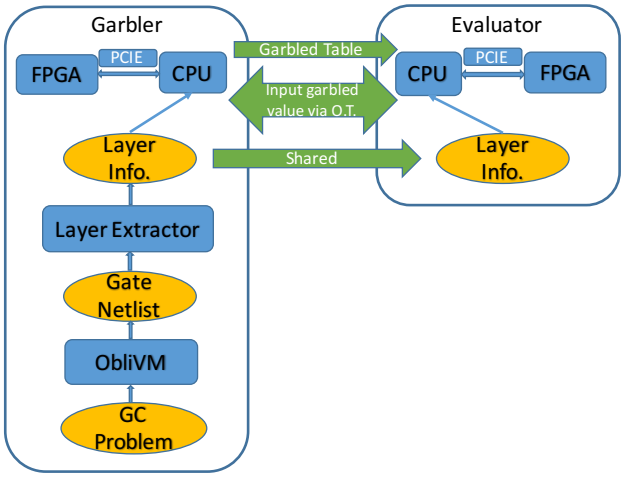


Figure 3: System Overview. An algorithm written in high-level language is translated to a Boolean circuit using ObliVM. The resulting circuit is passed through a layer extractor, identifying layers through BFS. The Boolean circuit DAG, annotated with layer information for each gate, is passed to the Garbler and Evaluator CPUs, that use it as a "blueprint" for execution. The CPUs subsequently use this blue print to garble gates and evaluate them at the FPGAs of the Garbler and Evaluator, respectively.

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 XORs: no ciphertexts need to be communicated between them for these gates. Our implementation takes advantage of all of these optimizations. We note that, as a result, the circuit for computing an AND gate, illustrated in Fig. 5, differs slightly from the AND gate garbling algorithm outlined above.

4. FPGA OVERLAY ARCHITECTURE FOR GARBLED CIRCUITS

4.1 System Overview

Our FPGA acceleration of GC works as follows. We start with a function f we wish to evaluate securely and a set of user inputs and generate an output of the function without revealing the data. This is done by (a) translating the function to a Boolean circuit and providing commands to the FPGA to garble/evaluate the function, and (b) accelerating the garbling or evaluation making use of an FPGA overlay architecture. We first describe how the system is partitioned between CPU and FPGA processing, and then describe the FPGA implementation.

Our system consists of two host PCs (instantiating the Garbler and Evaluator, respectively) equipped with FPGA cards for acceleration, as shown in Figure 3. On the Garbler side, the function f is first translated to a Boolean circuit before it can be garbled. Toward this end, we make use of ObliVM [22], a software framework that allows developers without any cryptography expertise to convert algorithms expressed in a high-level language to GCs. A user writes their problem in Java, and ObliVM translates it to a Boolean circuit and handles the garbling and evaluation. On the garbler, we take the Boolean circuit representation output from ObliVM, disabling garbling and evaluation through the framework. The Boolean circuit is sent to our layer extractor: this program extracts the Boolean circuit's *layers* using breadth first search. The resulting layered Boolean circuit (i.e., the "blueprint" for the garbling and execution) is sent to the Evaluator. This "blueprint" is subsequently used by the two hosts CPUs to dictate the garbling and evaluation to be performed by each FPGA.

In more detail, on each CPU, the Boolean circuit is represented as a DAG whose nodes are AND and XOR gates. Layers are defined recursively: gates whose input wires are global inputs are at layer 0, while a gate is at layer k if one of its input wires connects to a gate at layer $k-1$ and the other connects to a gate at layer $\leq k-1$. Note that gates in the same layer can be executed (i.e., garbled or evaluated) in parallel. Layer information is used to guide the CPU on the order with which gates are to be loaded to the FPGA, to be gabled or evaluated.

The FPGA implements a sea of garbled AND gates and XOR gates as described in Section 4.2. Each wire of the Boolean circuit has a unique wire ID associated with it. These wire IDs are also used as the memory addresses on the FPGA: these memory locations store the keys corresponding to these wires, used to garble or evaluate a gate. The CPU is responsible for mapping Boolean circuit gates to the FPGA hardware AND and XOR gates that realize them. For XOR gates this is trivial, since we implement one XOR gate in hardware (see below). AND gates are a different matter, as our FPGA architecture implements as many AND gates in parallel as can be kept busy. Suppose the FPGA realizes A AND gates in hardware. If there are more than A AND gates in a layer, our FPGA architecture is designed in such a way that the first hardware AND gate will complete processing before the information for the $A+1$st garbled gate is received by the FPGA. Thus, the CPU can transmit all the AND gates in a layer, and assign them to gates modulo A. More details of the FPGA architecture are given in Section 4.2.

When a layer is completed, the CPU then transmits the next layer of the circuit, until the circuit has been fully garbled. The CPU determines the order to send AND and XOR gates to the FPGA. Currently we send all the AND gates followed by all the XOR gates in a layer. Since the latency of an AND gate is much longer (82 cycles) than the latency of an XOR (one cycle), this results in a relatively efficient ordering.

For each layer, the Garbler sends to the FPGA (a) the number of gates in the layer, (b) the input and output wire IDs for the layer, (c) labels indicating whether a gate is AND or XOR, and (c) for the AND gates, which hardware gate the AND is mapped to (among the A available gates). Layer 0 requires key values for the inputs, which are 80 bit random values generated for each possible input value, i.e., $k^0_{w_i}$ and $k^1_{w_i}$. These strings are generated using a random number generator on the host for each of the input wires w_i and communicated to the FPGA. The output of the FPGA garbling includes the ciphertext values (i.e., the garbled gates), as well as keys for output wires. These are sent by the FPGA to the host CPU; the Garbler CPU sends garbled gates directly to the Evaluator CPU. The Garbler CPU also provides input keys to the Evaluator CPU via proxy oblivious transfer

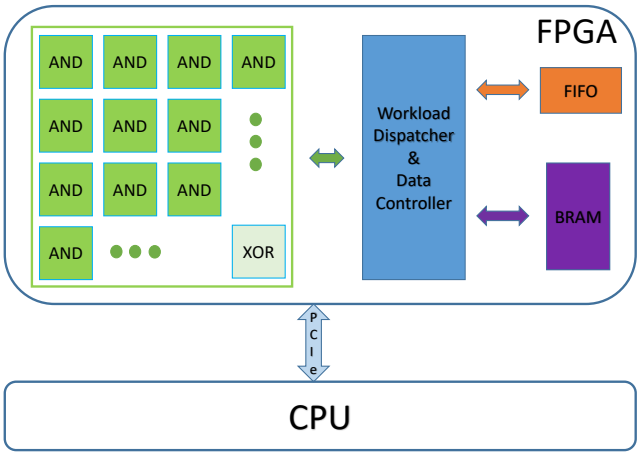


Figure 4: Overlay Architecture for Garbled Circuits. The overlay consists of as many AND gates as can be kept busy in parallel, a single XOR gate, BRAM for storing garbled wire values, and a FIFO for communicating values from the FPGA to the CPU. The workload dispatcher coordinates all operations.

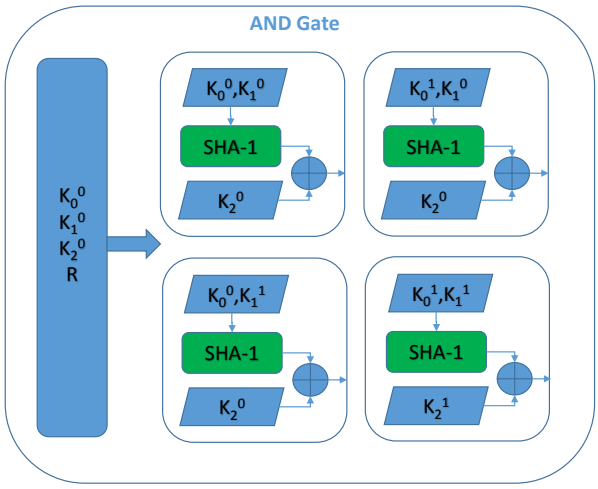


Figure 5: A Garbled AND Gate. The garbling of an AND gate consists of four SHA-1 cores operating in parallel.

between the Garbler, Evaluator, and the users/input owners, as described in Section 3.1.

4.2 FPGA Overlay Architecture

Our FPGA overlay architecture differs from other overlays in that it is designed to only support garbled circuits, whose implementation consists entirely of AND and XOR gates. Note that the overlay for the garbler and the evaluator are different. We support communication between gates by storing all inputs and outputs in on-chip block RAM. This is a coarse grained overlay, as both AND gates and XOR gates are quite complex, as described below.

A big advantage of implementing the garbler (and evaluator) as an overlay architecture is that it eliminates the lengthy place and route times incurred when using an FPGA. Different pieces of the same problem, as well as different problems, can easily be mapped to the overlay without incurring this expense. As the garbler is more complex than the evaluator, the rest of the paper describes the garbler and its implementation in detail.

The complete design of the overlay architecture for garbling, shown in Figure 4 includes XOR and AND gates, BRAM, a FIFO for communicating the garble table and outputs with the CPU, and a workload dispatcher. We describe these components and their design in this section.

BRAM. We use Block Random Access Memory (BRAM) to store the garbled values for each wire (i.e., every input and output for every gate). We treat all of the on-chip memory as one monolithic sequential memory device. The memory is 81 bits wide (80 bits of data plus a valid bit), and implemented with one read port and one write port. The unique wire IDs in the circuit, generated on the host CPU, correspond to memory locations. The maximum number of wire IDs that BRAM can hold is 2^{21}, assuming all memory locations are used for garbled values. This monolithic memory simplifies our design since no decision making is required in determining where to find inputs or where to store outputs. It also means that the BRAM is the current bottleneck in our design.

and and xor gates. We stress that the AND gates and XOR gates required for garbling are much more complicated than single bit gates. Inputs to all gates are represented as 80 bits in our implementation. Thus, a so called "free" XOR gate consists of eighty single bit XORs.

A garbling AND gate implements the functionality described in Section 3.1 and shown in Fig. 2. Each line is implemented according to Equation 1. This implementation requires four SHA-1 cores, using 512 bits of input derived from the garbled inputs and additional information. The implementation is based on an open source SHA-1 core [39]. Our garbled AND gate requires 82 clock cycles on the FPGA and uses 3070 ALMs and 3750 one bit registers on a Stratix V FPGA. Note that, while SHA-1 in and of itself is no longer considered secure, it is adequate for preserving privacy in the context of garbled circuits, where cryptography is applied at many levels. In addition, new garbled values and keys are generated whenever new input values are applied to a circuit, making any attack unlikely to succeed. Figure 5 shows the implementation of four SHA cores in parallel in our design.

We implement in hardware the maximum number of AND gates that can be kept busy, taking the latency of the AND operations and the availability of the BRAM for reading into account. In our current design, this results in $A = 43$ AND gates. We implement a single XOR gate, as the computation has one cycle latency and additional XORs cannot be provided with inputs. Fig. 6 shows the timeline for garbling a circuit consisting of only AND gates. Our overlay architecture implements 43 AND gates. If more ANDs are in a layer the 44th gate runs on the first gate when it is completed. In our implementation, the XOR gates in a layer will be computed after all the AND gates have started. An XOR gate has four cycles of latency total, two for reading inputs, one for computing XOR and one for writing the output. There may be contention for writing BRAM if XOR and AND gates complete at the same time. This contention is handled by the workload dispatcher.

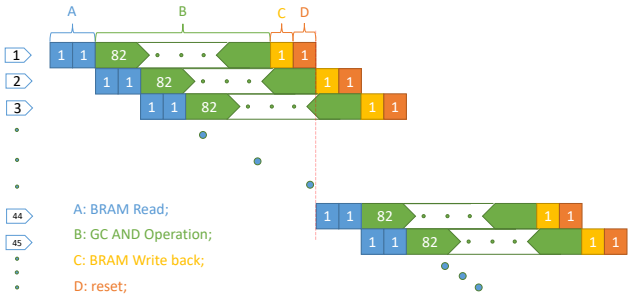


Figure 6: Garbled Circuit AND Gate Operation Sequence. Reads and writes from BRAM are the bottleneck, since BRAM has one read and one write port. An AND gate can begin operation after its two (80 bit) inputs are read from BRAM. Each AND gate has 82 clock cycle latency. This diagram assumes there are no XOR gates. Latency for processing XOR gates is much shorter.

FIFO. The FIFO enables communication from the FPGA to the CPU; data is sent over the PCIe bus. The values transmitted include the ciphertexts for each AND gate and the keys of each gate's (AND and XOR) output wires. The garble table values are written to the FIFO when each AND garbling completes. The FIFO is wide enough for this to take one clock cycle. This is done during the "reset" cycle shown in Fig. 6. At the completion of garbling the output wire values are also written to the FIFO.

Workload Dispatcher. The workload dispatcher is a state machine that receives input from the host, reads and writes the correct values from BRAM, and properly dispatches outputs and the garble table values. Specifically, the workload dispatcher implements the following steps:

1. Determine if next gate to be processed is an AND or XOR gate.
2. Read the inputs, and forward to the assigned gate (recall that AND gates are assigned by the host).
3. When a gate is finished computing, write the output of the garbled gate to the correct location in BRAM.
4. When an AND gate is finished computing, push the gate ID and garble table values to the FIFO for transmission to the host PC over the PCIe interface.
5. At the end of garbling, read the garbled output value(s) from the wire ID(s) provided by the CPU and push them to the FIFO for transmission to the CPU.

5. EXPERIMENTAL RESULTS

We use the ProcV board from Gidel as our target platform. It is a Stratix V FPGA-based platform with 16+GB external memory. It provides high-speed communication between host and FPGA via a PCIe*8 generation 3 bus which makes the system suitable for high-performance computing and low latency networking projects. The ProcV system is supported by Gidel's ProcWizard software and IPs. The Altera Stratix V FPGA on board provides high capacity and high speed for many designs and contains 234K ALMs and 52M memory bits.

Table 1: Resource Utilization

Module	ALMs	M20Ks	1 bit Register
One AND	3,070	0	3,750
One XOR	40	0	81
BRAM	0	1,060	0
FIFO	510	280	404
Whole Design	176,893/234,720 75.4%	1,340/2,560 52.3%	215,308

PCIe generation 3 is a serial computer bus standard which has been available since 2010. It doubles the data rate compared with generation 2 with 8 Giga transfers per second(GT/s) per lane. The ProcV board, with 8 lanes, will provide about 7.88 Gb/s throughput. This high throughput benefits the data transfer between the host CPU and the Stratix FPGA on board. For garbled circuits the amount of data to transfer is not high, but having high speed interconnect ensures that communication between FPGA and host is not the bottleneck.

Table 1 shows the resource utilization for our system. For this system, the total logic utilization (in ALMs) is about 75% of those available. The BRAM can hold 2^{21} words, where a word is a garbled value. Any garbled circuit problem with fewer than 2^{21} wire IDs can fit in our system.

We use ObliVM [22] to generate the Boolean circuit representation fed into the FPGA. We also use it to run our experiments to completion. We compare our results with ObliVM to validate our designs and also compare run times to show speed up. Our design is not fully working in hardware so the experimental results we provide are estimates based on the design tools and placed and routed circuits. The maximum frequency achievable for this overlay architecture is just over 200 MHz.

We compare the number of clock cycles for both ObliVM garbling a circuit and our approach. ObliVM is written in Java; we insert some C code which can precisely monitor the clock cycle count. We sum the clock cycle times for the XORs and ANDs to provide the computing time on both the FPGA and host CPU. Note that these operations are performed serially on the CPU, but in parallel on the FPGA. We do not include some of the setup time in ObliVM. A complete end-to-end test should show an even greater advantage for GC on FPGAs.

The problems that we garble are: Millionaire's problem, addition, Hamming Distance (HD), multiplication and sorting. The size of these problems is shown in Table 2. For different problems we use different numbers of input bits. The millionaire's problem uses 2 bits for each person; the adder is 6 bits wide. Sorting orders a sequence of inputs; in this example the inputs are ten four bit integers. In addition to explore scalability, we implement several different sizes of HD and multiplication. For HD, we show results for two 10, 20 and 30-bit inputs. For multiplication, we show results for 8, 16, 32 and 64 bit multipliers. Table 2 shows the number of AND and XOR gates for each of these problems, as well as the number of layers and maximum number of AND gates per layer.

Table 3 compares clock cycles for our FPGA design and ObliVM software. The FPGA implementation requires about 10^4 times fewer clock cycles than ObliVM, and demonstrates the advantage of implementing garbling using FPGAs. The

Table 2: Size of the Examples

Problem	# of AND Gate	# of XOR gate	# of layers	Max # of AND gate in One Layer
Millionaire (2)	2	11	7	1
Addition (6)	6	24	18	1
Hamming Distance (10)	20	90	22	5
Hamming Distance (30)	60	270	28	15
Hamming Distance (50)	100	450	33	25
A * B (8)	120	352	57	64
A * B (16)	496	1472	122	256
A * B (32)	2016	6016	250	1024
A * B (64)	8128	24320	505	4096
Sorting (10*4)	848	4683	278	32

Table 3: Clock Cycle Comparison

Problem (Input Size in bit)	Our Approach (Clock Cycle)	ObliVM (Clock Cycle)
Millionaire (2)	$1.9 * 10^2$	$1.1 * 10^6$
Addition (6)	$5.6 * 10^2$	$1.7 * 10^6$
Hamming Distance (10)	$1.2 * 10^3$	$4 * 10^6$
Hamming Distance (30)	$2.2 * 10^3$	$1.1 * 10^7$
Hamming Distance (50)	$2.8 * 10^3$	$1.7 * 10^7$
A * B (8)	$4.4 * 10^3$	$3 * 10^7$
A * B (32)	$3.6 * 10^4$	$1.1 * 10^8$
A * B (64)	$1.1 * 10^5$	$3.2 * 10^8$
Sorting (10*4)	$1.1 * 10^4$	$1.4 * 10^8$

Table 4: Real Time Speedup

Problem	Speedup
Millionaire (2 bits)	422
Addition (6 bits)	222
Hamming Distance (10 bits)	243
Hamming Distance (30 bits)	357
Hamming Distance (50 bits)	434
A * B (8 bits)	498
A * B (32 bits)	218
A * B (64 bits)	208
Sorting (10*4 bits)	929

software platform runs on a computer with Intel Core i7-2640 CPU at 2.80 GHz; the FPGA design runs at 200 MHz. Taking this into consideration, Table 4 presents the expected speedup of our approach compared with ObliVM. Our approach is two to three orders of magnitude faster. In addition, we expect the FPGA implementation to consume much less power. Runtime results for HD show that the speed up increases with the size of the inputs. For multiplication, we see the opposite trend with speedup decreasing as the problem gets larger. This is partly due to the fact that the large multiplication examples have more layers, and some of these layers have very few AND gates. In our design, much of our parallelism, and therefore speedup, comes from processing multiple garbling AND gates in parallel. In future work, we will examine partitioning schemes other than "layer by layer" which should allow us to take better advantage of the available parallelism.

This project has two goals. The first is to accelerate garbled circuits on an FPGA. The second is to be able to rapidly garble different problems without incurring long recompile and programming costs. Compiling a garbled circuit design from scratch could take hours or even days. Our overlay architecture is compiled and downloaded once. This initial time is about an hour for the current design. The time to change between problems on the FPGA is very small. The FPGA design needs to be reset to clear all sequential elements, and then new values can be transmitted. For the small problems evaluated here, the longest step for garbling a compete function is the time to translate a problem to a Boolean circuit. This time can be minimized by keeping a library of garbled circuit implementations available, essentially caching the output of the ObliVM circuit generator for future use.

6. CONCLUSIONS AND FUTURE WORK

We have presented an implementation of Garbled Circuits on an FPGA using a coarse-grained overlay architecture. Our implementation is the first FPGA overlay architecture designed for this purpose, makes use of the parallelism available on the FPGA to accelerate garbling, and can achieve more than two orders of magnitude speedup over existing software implementations. This advantage is due to the fact that the operations encountered in GCs, (well defined primitive operations involving key encryptions, hashing and XOR computation) are a good match for an FPGA architecture implemented as an overlay. The design is demonstrated on small problems, but is designed to scale up to much larger ones. Larger problems can be divided into partitions and each part can be sequentially implemented on the FPGA overlay. The overhead for switching between parts of the same problem or between different problems consists of a few words of information communicated over a high speed PCIe interface.

In the future, we plan to examine much larger GCs, and expect the advantage of the FPGA implementation to grow as the problem size increases. We will implement the host code to partition and schedule garbled circuits onto the FPGA fabric and consider problems that map to multiple FPGAs on multiple nodes in a data center setting. Our architecture is the initial step to making privacy preserving computation on a large scale feasible by accelerating it with FPGAs.

Acknowledgments

This research was supported by Google through a Faculty Research Award.

References

[1] L. A. Adamic and N. Glance. The political blogosphere and the 2004 US election: divided they blog. In *Proceedings of the 3rd international workshop on Link discovery*, pages 36–43. ACM, 2005.

[2] J. Angwin. The web's new gold mine: Your secrets: a journal investigation finds that one of the fastest-growing businesses on the internet is the business of spying on consumers; first in a series. *Wall Street Journal*, 2010.

[3] 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*, pages 503–513. ACM, 1990.

[4] M. Beye, Z. Erkin, and R. L. Lagendijk. Efficient privacy preserving k-means clustering in a three-party setting. In *IEEE International Workshop on Information Forensics and Security*. IEEE Press, 2011.

[5] A. Brant and G. G. Lemieux. ZUMA: An open FPGA overlay architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 93–96. IEEE, 2012.

[6] W. Du, Y. S. Han, and S. Chen. Privacy-preserving multivariate statistical analysis: Linear regression and classification. In *4th SIAM International Conference on Data Mining*. SIAM, 2004.

[7] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6), 1985.

[8] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge university press, 2009.

[9] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In *ACM Conference on Computer and Communications Security*, 2010.

[10] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.

[11] Y. Huang, L. Malka, D. Evans, and J. Katz. Efficient privacy-preserving biometric identification. In *Network and Distributed System Security Symposium*, 2011.

[12] N. Husted, S. Myers, A. Shelat, and P. Grubbs. GPU and CPU parallelization of honest-but-curious secure two-party computation. In *Asia-Pacific Computer Science and Application Conference*, 2013.

[13] A. K. Jain, S. A. Fahmy, and D. L. Maskell. Efficient overlay architecture based on DSP blocks. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 25–28. IEEE, 2015.

[14] A. K. Jain, D. L. Maskell, and S. A. Fahmy. Are coarse-grained overlays ready for general purpose application acceleration on FPGAs? In *Proceedings of IEEE International Conference on Pervasive Intelligence and Computing*. IEEE, 2016.

[15] K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. In *Cryptographic Hardware and Embedded Systems*, pages 383–397. Springer, 2010.

[16] N. Kapre and J. Gray. Hoplite: Building austere overlay NoCs for FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 1–8. IEEE, 2015.

[17] N. Kapre, N. Mehta, M. Delorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. Dehon. Packet switched vs. time multiplexed FPGA overlay networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 205–216. IEEE, 2006.

[18] D. Koch, C. Beckhoff, and G. G. Lemieux. An efficient FPGA overlay for portable custom instruction set extensions. In *International Conference on Field Programmable Logic and Applications*, pages 1–8. IEEE, 2013.

[19] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *International Colloquium on Automata, Languages and Programming*, 2008.

[20] M. Kosinski, D. Stillwell, and T. Graepel. Private traits and attributes are predictable from digital records of human behavior. *Proceedings of the National Academy of Sciences*, 110(15):5802–5805, 2013.

[21] J. Leskovec, L. Backstrom, and J. Kleinberg. Meme-tracking and the dynamics of the news cycle. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 497–506. ACM, 2009.

[22] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. Oblivm: A generic, customizable, and reusable secure computation architecture. In *IEEE Symposium on Security and Privacy*, 2015.

[23] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay-secure two-party computation system. In *USENIX Security*, volume 4, 2004.

[24] A. Mislove, B. Viswanath, K. P. Gummadi, and P. Druschel. You are who you know: Inferring user profiles in Online Social Networks. In *International Conference on Web Search and Data Mining*, 2010.

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

[26] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy*, 2008.

[27] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: Parallel secure computation made easy. In *IEEE Symposium on Security and Privacy*, 2015.

[28] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. Privacy-preserving matrix factorization. In *ACM Conference on Computer and Communications Security*, 2013.

[29] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *IEEE Symposium on Security and Privacy*, 2013.

[30] J. Otterbacher. Inferring gender of movie reviewers: exploiting writing style, content and metadata. In *Conference on Information and Knowledge Management*, 2010.

[31] S. Pu, P. Duan, and J.-C. Liu. Fastplay-a parallelization model and implementation of SMC on CUDA based GPU cluster architecture. *IACR Cryptology ePrint Archive*, 2011:97, 2011.

[32] M. O. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Aiken Computation Laboratory, Harvard University, 1981.

[33] M. Ramos-Casals, P. Brito-Zerón, B. Kostov, A. Sisó-Almirall, X. Bosch, D. Buss, A. Trilla, J. H. Stone, M. A. Khamashta, and Y. Shoenfeld. Google-driven search for big data in autoimmune geoepidemiology: Analysis of 394,827 patients with systemic autoimmune diseases. *Autoimmunity reviews*, 2015.

[34] D. Rao, D. Yarowsky, A. Shreevats, and M. Gupta. Classifying latent user attributes in twitter. In *2nd International workshop on Search and mining user-generated contents*, 2010.

[35] S. Ressler. Social network analysis as an approach to combat terrorism: past, present, and future research. *Homeland Security Affairs*, 2(2):1–10, 2006.

[36] S. Salamatian, A. Zhang, F. du Pin Calmon, S. Bhamidipati, N. Fawaz, B. Kveton, P. Oliveira, and N. Taft. How to hide the elephant-or the donkey-in the room: Practical privacy against statistical inference for large data. In *2016 IEEE Global Conference on Signal and Information Processing*, 2013.

[37] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *IEEE Symposium on Security and Privacy*, 2015.

[38] 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 *Proceedings of the 53rd Annual Design Automation Conference*, page 73. ACM, 2016.

[39] J. Strömbergson. SHA1 core. https://github.com/secworks/sha1.

[40] U. Weinsberg, S. Bhagat, S. Ioannidis, and N. Taft. Blurme: Inferring and obfuscating user gender based on ratings. In *ACM conference on Recommender systems*, 2012.

[41] T. Wiersema, A. Bockhorn, and M. Platzner. Embedding FPGA overlays into configurable systems-on-chip: ReconOS meets ZUMA. In *2014 International Conference on ReConFigurable Computing and FPGAs*, pages 1–6. IEEE, 2014.

[42] J. Wortham. Facebook and privacy clash again. *The New York Times May*, 6, 2010.

[43] A. Yao. How to generate and exchange secrets. In *Foundations of Computer Science*, 1986.

[44] A. C.-C. Yao. Protocols for secure computations. In *IEEE Symposium on Foundations of Computer Science*, volume 82, pages 160–164, 1982.