Hindawi
Security and Communication Networks
Volume 2020, Article ID 3042642, 10 pages
https://doi.org/10.1155/2020/3042642

Research Article

Private Function Evaluation Using Intel's SGX

Omar Abou Selo,[1] Maan Haj Rachid ,[2] Abdullatif Shikfa,[1,3] Yongge Wang,[4] and Qutaibah Malluhi[1]

[1]*Qatar University, Doha, Qatar*
[2]*Karolinska Institutet, Solna, Sweden*
[3]*College of the North Atlantic, Doha, Qatar*
[4]*University of North Carolina, Charlotte, NC, USA*

Correspondence should be addressed to Maan Haj Rachid; maan.rachid@scilifelab.se

Received 25 September 2019; Revised 25 August 2020; Accepted 28 August 2020; Published 15 September 2020

Academic Editor: Flavio Lombardi

Private Function Evaluation (PFE) is the problem of evaluating one party's private data using a private function owned by another party. Existing solutions for PFE are based on universal circuits evaluated in secure multiparty computations or on hiding the circuit's topology and the gate's functionality through additive homomorphic encryption. These solutions, however, are not efficient enough for practical use; hence there is a need for more efficient techniques. This work looks at utilizing the Intel Software Guard Extensions platform (SGX) to provide a more practical solution for PFE where the privacy of the data and the function are both preserved. Notably, our solution carefully avoids the pitfalls of side-channel attacks on SGX. We present solutions for two different scenarios: the first is when the function's owner has an SGX-enabled device and the other is when a third party (or one of the data owners) has the SGX capability. Our results show a clear expected advantage in terms of running time for the first case over the second. Investigating the slowdown in the second case leads to the garbling time which constitutes more than 60% of the consumed time. Both solutions clearly outperform FairplayPF in our tests.

1. Introduction

In Private Function Evaluation (PFE), a participant S_0 holds some private function f, while participants $S_1, S_2, \ldots, S_m$ each have their own private input x_i. These parties would like to work together to find $f(x_1, x_2, \ldots, x_m)$ while retaining the confidentiality of their respective inputs and of S_0's function.

This problem is useful when an entity holding a proprietary piece of software would like to offer some service using that software to other entities that have confidential data. One typical example would be a privacy-centric recommendation system. Using PFE, a company can run their proprietary algorithms to recommend products to consumers while maintaining privacy of consumers data even from the company itself.

This problem is similar to Secure Multiparty Computation (SMPC) in that both problems require the input data to remain hidden. However, PFE additionally requires the function to be private while SMPC assumes a publicly known function. The performance of SMPC solutions has improved a lot over the years making SMPC more practical and thereby more widely adopted. This is not the case with PFE as the additional requirement of function privacy adds more complexity to the problem. Although not very practical, solutions for PFE do exist and are mostly adapted from techniques used in SMPC.

One such solution involves running a universal circuit in SMPC that takes $x_1, x_2, \ldots, x_m$ in addition to C_f, a circuit representation of f, as inputs. The idea is that SMPC insures the privacy of all inputs; hence the privacy of the function is insured since it is part of the input. The issue, however, is that a universal circuit that can run C_f will be of size $\Omega(|C_f| \log |C_f|)$ according to the state of the art [1]. Different universal circuits must be constructed depending on $|C_f|$ adding further cost to this solution.

More recent solutions involve modifying the garbled circuits used in SMPC in order to hide the gates functionality and circuit topology of C_f. These approaches can actually achieve a linear cost of $|C_f|$, but with an additional cost of a linear amount of asymmetric key operations which are not practical. Without asymmetric key operations, the best solution still takes $O\ (|C_f|\ \log |C_f|)$ time. Hence, these approaches are also not very practical.

In this article, we propose a practical solution to the PFE problem which builds on Intel Software Guard Extensions (SGX). SGX is a set of CPU instructions provided by Intel to allow hardware-based protection of the running software. SGX provides this protection through the use of enclaves, which are protected areas of execution in memory. Code and data in an enclave will be encrypted and hidden from even the operating system. SGX provides powerful tools that can help in developing a more practical PFE solution. However, we note that SGX does not hide memory accesses and instruction trace; hence it is vulnerable to side-channel attacks which we should carefully consider.

This work looks at developing protocols that make use of SGX to create more practical solutions for PFE.

Our key contributions can be summarized as follows:

(i) We propose the first approach built on SGX to offer efficient PFE.

(ii) We consider two scenarios for our solution: one where the function owner has the SGX-enabled device and a more challenging scenario where a data owner or any third party has SGX that assists in performing PFE.

(iii) We analyze the security and the performance of our scheme theoretically. We also implement a proof-of-concept of our solutions in both scenarios to benchmark the efficiency of our approach and show that it outperforms current existing solutions.

2. Preliminary Definitions and Background

In this section, we present an overview about concepts and technologies which are needed to build our solutions.

2.1. SGX. SGX is a new set of instructions which were provided by Intel in order to enable developers to create enclaves which are protected areas in memory. SGX guarantees both integrity and confidentiality to data and code inside the enclave. Intel provided an API for developers to create applications with enclaves. SGX supports multiple enclaves with a limited total size of 128 MB. Unfortunately, SGX does not hide the memory access of the enclave which opens the door for both deterministic and probabilistic side-channel attacks [2].

Remote SGX attestation provides a hardware-based guarantee that a certain software is running on another server's enclave. This means that a client will be able to gain confidence that the server it is communicating with is running a legitimate enclave. Attestation also provides a secure communication channel by establishing shared keys between the client and the enclave which enables the client to encrypt messages that can only be read by the enclave itself but not by the host of the enclave.

2.2. Garbled Circuits (GC). Originally introduced by Yao [3], GC is a technique that performs secure multiparty computation in the two-party setting. A formal description of GC can be found in [4].

In GC, the function is assumed to be public and represented by a circuit. The two parties work together to evaluate the result of the function on their respective private inputs. One party assumes the role of garbling the circuit and is denoted by the garbler, while the other party evaluates the garbled circuits and is referred to as the evaluator.

Garbling means blurring the circuit in a way, and it implies replacing the wire values which are 0 or 1 prior to garbling to pseudorandom values of a size defined by the security parameter k. For each wire w_i, the garbler picks two random k-bit values w_i^0 and w_i^1 to denote its new 0 and 1 values. For each gate whose input wire labels are w_l and w_r, output wire label is w_o, and truth table is T; the garbler replaces T with the garbled table GT such that if $T\ (a,\ b) = c$, then $GT\ (w_l^a, w_r^b) = E_{w_l}(E_{w_r}(w_o^c))$.

After garbling, the garbler sends the garbled circuit and his garbled inputs to the evaluator. The evaluator must also receive his garbled input representation from the garbler. This is done through Oblivious Transfer (OT). OT is a method that allows the garbler to present two choices w_i^0 and w_i^1 to the evaluator and the evaluator has to pick one of these two choices without learning the other choice and without allowing the garbler to learn the choice which was picked by the evaluator.

Given all garbled input and garbled circuit, the evaluator evaluates the garbled gates in topological order until the final output is reached. Each garbled gate is evaluated by decrypting each ciphertext in its garbled table using the two gate inputs w_l^a and w_r^b and checking if the decrypted value is a valid gate output (could be done by padding the valid gate output with zeros). The final output of the circuit can be sent back to the garbler to decode it.

2.3. GC Optimizations. There have been many presented optimizations for GC that greatly improved its practicality. These optimizations generally follow two different directions:

(i) Reducing the number of ciphertexts per gate in order to reduce network transfer time [5–8]

(ii) Improving the computational efficiency of garbling and the evaluation of the circuit [9, 10]

The first direction involves a specific process of generating the garblings which, as a consequence, creates a dependency between gates' input wires and the corresponding output wire. These dependencies do not leak information and are proven secure.

The second direction for optimizations aims to make the garbling and evaluation computation more efficient without paying any cost in terms of the security of the overall garbled circuits. This work makes use of two of these optimizations: Point and Permute and Fixed-key block cipher.

Point and permute [10] assigns a bit to each wire garbling and permutes the truth table in a way that makes it possible for the evaluator to realize which ciphertext from the truth table he should be decrypting, thereby removing the need of further decryption to make sure that the correct one is decrypted.

Fixed-key block cipher [9] notes that fixing the key used during garbling and evaluating makes the block cipher operation a simple permutation that is executed much faster than evaluating the full block cipher since the computations that only depend on the key can be precomputed before hand.

2.4. Universal Circuits. A universal circuit UC is a circuit that takes other circuits as input and evaluates them. If C is some circuit that UC supports and x is some input to C, then, UC $(C, x) = C(x)$. The main issue is the size of the universal circuit, and recently researchers have proposed solutions to obtain smaller universal circuits. For the class of size n circuits, Valiant's universal circuit [11] is of size 19 $n\log n$ with depth $O(n)$ and Kolesnikov and Schneider's universal circuit [1] is of size 1.5 $n\log^2 n$ though it has smaller universal circuits for circuit sizes less than 5000. Kiss and Schneider [12] further reduced Valiant's bound by constructing universal circuit where the number of AND gates is bounded by 5 $n\log n$ and where the number of total gates is bounded by 20 $n\log n$. Although Kiss and Schneider [12] showed that it is practical to implement PFE using Valiant's size-optimized universal circuits, they claimed that "*universal circuits are not the most efficient solution to perform PFE.*" Despite the fact that implementations for Secure Function Evaluation (SFE) protocol with billions of gates have been reported in the literature, the best reported implementation for a universal circuit based PFE protocol [12] is for simulated circuits of 300,000 gates, which results in a universal circuit of at most 245,627,140 gates (at most 61,406,785 AND gates).

2.5. Terminology. We define the "circuit owner" as the party who has a private function, while a "data owner" is a party who owns private data. The terms program owner and the circuit owner are used interchangeably. The "enclave parent" is the party that creates an enclave in which all sensitive computations are done. We also use the terms enclave parent and the enclave host interchangeably.

3. Related Work

In this section, we review related work on secure multiparty computations and give an overview for the development of garbled computing. We also discuss SGX applications and the advantages of using SGX in solving PFE.

3.1. GC with Universal Circuits. GC provides secure multiparty computation (SMPC) and hides the input data but assumes a public function. If, additionally, the function that the program owner wishes to evaluate is passed as part of the input to SMPC, then the function will be hidden together with the input data. Thus, one way to achieve PFE is by running a UC using GC whose input is C_f and x. Note that any kind of GC optimization is applicable here because it is not required to hide the functionality of the UC (which is actually publicly known). However, there will be an additional logarithmic cost incurred since a UC will be of size Ω $(|C_f| \log |C_f|)$. Additionally, a UC has to be constructed specifically to run a certain set of possible circuits. If the UC construction cost was pushed to offline, then the produced UC needs to be big enough to evaluate any possible circuit input which will create a very big UC. Nevertheless, this approach does have some benefits as it supports multiple data owners. The current state-of-the-art UC construction for small circuit was proposed by Kiss and Schneider in [12] and achieves an upper bound of 1.5 $n\log^2 n$. One may also use TinyGarble [13] techniques to construct smaller circuit for PFE. Most existing garbled circuit techniques convert a function/program to a combinational Boolean function with a directed acyclic graph (DAG) of binary gates. The authors in [13] analyzed the approach which first converts a function/program to a sequential circuit, which allows having feedback from the output to the input by adding the notion of a state (memory). Then, one can convert each sequential cycle to a Boolean combinational logic.

The results in [13] show that this approach can reduce the size of the garbled circuit significantly. FairplayPF [1] is a well-documented framework for secure evaluation of private functions using universal circuits. It is an extension of the classical Fairplay [14], which is a tool for secure two-party computation with a publicly known function. Most of the proposed PFE techniques have been of theoretical interest. They lacked implementation and lacked tools for program (private function) development. This is attributed to the fact that PFE is still very slow to provide performance that meets the time requirements of real applications. FairplayFE was unique in the sense that it provided a tool and an implementation of PFE. Therefore, we used FairplayFE as the baseline to evaluate the performance of our proposed techniques.

3.2. Modified GC: Nonuniversal Circuit-Based PFE. In the modified GC, we try to hide both the individual gate functions and the topology of the circuit C_f. The gate functions can be hidden by using only universal gates such as the NAND. Indeed the result will be that all gates have the same function which does not leak information. Hiding the topology is a bit more tricky though. To do so, existing solutions make a distinction between outgoing wires and ingoing wires. Outgoing wires are gate output wires and the circuit input wires. Assuming that the input size is n, then the number of outgoing wires is $n + |C_f|$. Ingoing wires are gate input wires which means that the number of ingoing wires is 2 $|C_f|$ (as we assume that all gates are NAND which are binary gates). Ingoing wires and outgoing wires have different wire labels and in order to connect an outgoing wire like a gate output to an ingoing wire, some translation needs to take place. This translation needs to be oblivious to the data owner but can be known to the program owner. There are currently two solutions that achieve this, one that uses additive homomorphic encryption [15] and another that uses switching networks [16].

Katz and Malka also introduced a more efficient variant PFE protocol with provable security in the random oracle model. The second protocol is roughly twice as efficient as the first one. PFE protocols in [15, 16] use a singly homomorphic public-key encryption scheme such as the additive homomorphic Paillier encryption scheme. Let C_f be a circuit that computes P_2's function f and that C_f contains only NAND gates. Assume that C_f have n gates and it take l-bit inputs. In a high level, the PFE protocol proceeds as follows.

(1) Given the pair (n, l), P_1 generates a sequence of $n + l$ pairs of labels which are encrypted using singly homomorphic encryption scheme and sent to P_2

(2) P_2 obliviously groups these labels in gates to form a circuit C_f using a linear transformation compatible with the singly homomorphic encryption scheme and sends the gates to P_1

(3) After decrypting the gates, P_1 produces a garbled circuit corresponding to the circuit C_f by garbling the n gates received from P_2 independently (and P_1 does not learn the circuit in this way)

(4) P_1 gives an encoded version of the input x to P_2 and P_2 evaluates the garbled circuit to obtain the circuit output $C_f(x) = f(x)$

The PFE protocols in [15, 16] were described with P_2 learning the output $f(x)$. The limitation of this approach is that the server P_2 is allowed to compute any function of his choice. Hence, a malicious P_2 can just provide a function to output the data owner inputs without violating the privacy definitions of the authors. For this reason, in our settings of the PFE protocols, we do not allow the function owner to get any output. Katz and Malka explain how to modify their protocol at no additional cost to achieve this. The modified version is the closest solution to our setting.

The PFE protocols in Katz and Malka [15] have provable security in the semihonest security model with the assumption of semantic security for homomorphic encryption schemes and linear-related key security for symmetric encryption schemes. It is also worth noting that Mohassel et al. proposed later a solution for PFE protocols which is secure against malicious adversaries [17]. The latter, however, relies heavily on zero-knowledge proofs and is therefore much more costly.

3.3. ORAM with GC. Based on the physically shielded Central Processing Unit (CPU) technique [18], Goldreich and Ostrovsky [19] proposed a theoretical treatment of software protection by formulating the problem in the setting of learning a program structure by observing its execution. Using this new formulation, they reduced this problem to online simulation of any programs on oblivious RAMs (random access machines). A machine is oblivious if its access to memory locations is independent of the input values and is processed with the same running time. Lu and Ostrovsky [20] showed how to design garbled ORAMs by constructing t pairs of garbled circuits $(O^i_{\mathrm{ORAM}}, O^i_{\mathrm{CPU}})$ for $i = 1, \ldots, t$, where t is the maximum runtime of the ORAM, O^i_{ORAM} simulates the ith-step memory read/write command, and O^i_{CPU} simulates the ith-step shielded CPU operation. Gentry et al. [21] showed that in order to prove the security for the garbled RAM scheme in [20], an additional circularity assumption is required. Gentry et al. [21] then proposed two new constructions to avoid this additional assumption.

3.4. SGX. SGX has been used in several applications such as private membership testing [2], oblivious RAM [22], and secure indexing [23]. SGX has also been used for secure multiparty computation. For example, Bahmani et al. [24, 25] designed practical secure multiparty computation (SMP) using SGX. As mentioned above, SGX may leak information about the program running through side-channel attacks. References [26, 27] use Intel SGX to achieve secure function evaluation, with the former trying to hide the function.

3.5. Benefit over Related Work. In both cases, using SGX can keep the cost linear to the $|C_f|$ which makes this solution more scalable to larger circuits in comparison with UC or the modified GC using switching networks. Compared to the nonuniversal circuit based PFE protocols in Katz and Malka [15] (see also [16]), our approach has two advantages. First, our approach allows more than one data owners to participate in the private function evaluation while the PFE in [15] has only two participants: one data owner and one circuit owner. Secondly, our approach only uses symmetric key ciphers while the PFE protocols in [15, 16] requires an additive homomorphic encryption scheme to obliviously connect each of the gates. That is, at least two extra additive homomorphic encryption operations for each wire are required for each participant. These additive homomorphic encryption operations are the major cost for the PFE schemes in [15, 16]. Thus, our scheme is significantly more efficient.

4. PFE Leveraging SGX

A trivial design of PFE with SGX may run a private program within an SGX enclave directly. One might think for example to run the program in an enclave and interpret it there. The attestation can show that the enclave is running a trusted interpreter (e.g., Java Virtual Machine), and keys to decrypt the data to perform the computation can be securely shared with the enclave and not with the developer of the private function. However, this approach is vulnerable to several attacks. For example, one may use the program's runtime or the memory access pattern to infer some information about private inputs (see, e.g., [25]). A number of research studies have actually shown that utilizing a cache can reveal details about the execution of a program [28–32].

To defeat these side-channel data leakage attacks that SGX is susceptible to, the program is represented as a circuit. This is because circuits perform the same memory accesses regardless of the input data and hence are memory access oblivious. Accordingly, the words "circuit" and "program" are used interchangeably.

While SGX is available on most modern Intel CPUs, it may still be the case that a machine does not have SGX support; hence we propose two solutions that cover different assumptions:

(i) The program owner has an SGX supported CPU. This is the simpler of the two cases.

(ii) The program owner's machine does not support SGX. In this case, SGX support is available either on the data owner's machine, or on the machine of an untrusted third party other than the data or program owners. The technique to address this second scenario is the same regardless of whether SGX is available on the machine of a third party, or the machine of the data owner (in this case, the data owner takes the role of the third party).

4.1. Technique 1: Circuit Owner Has SGX. In the first case, the circuit owner creates an enclave with a specific list of tasks. This enclave must first attest to the data owners that it is a valid enclave and that it is performing the right protocol. For example, the enclave should attest to the data owners that the enclave will only issue Ocall at the end of the function evaluation and the Ocall should only return encrypted output value under an established key with the data owner. Furthermore, the enclave should also attest to the data owners that the enclave will only accept circuit description (instead of general program descriptions with loops, etc.) from the program owner. During the attestation phase, a secret key between each data owner and the enclave will be established using the Diffie–Hellman key agreement protocol. This key will be used to encrypt all future communication content between the data owner and the enclave. Specifically, the data input will be encrypted using this key. This concludes the initialization phase. Starting the online phase:

(i) The data owners encrypt their input using the established secret keys and send it to the enclave via the enclave's parent.

(ii) The enclave also receives the circuit from the program owner who is also its host (parent).

(iii) Since both data and circuit are in the enclave, it performs the evaluation and obtains an output. The output is encrypted and then sent to the data owners so that only they can learn the output and not the program owner.

Figure 1 demonstrates case 1. Data owner and enclave are trusted parties while program owner who is the enclave's parent is an untrusted party.

Note that this case is simpler than the next one, as the function does not need to be private with regard to the enclave. The function is indeed executed at the function owner side, but in an enclave, and the function owner naturally knows the function.

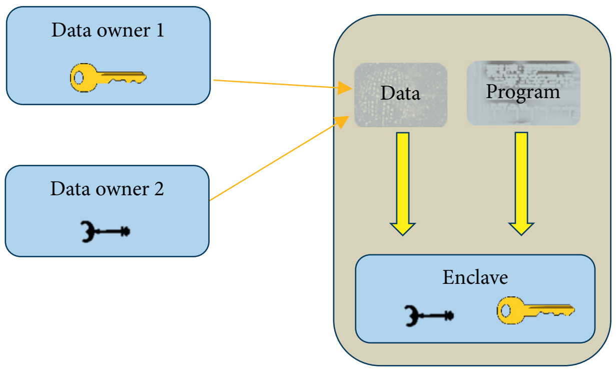

Figure 1: Technique 1. The program owner which is the enclave's parent is an untrusted party.

4.2. Technique 2: The Circuit Owner Does Not Have SGX. In the second case, the circuit owner does not have an SGX but a data owner or a third party has an SGX-enabled hardware. In the following, we consider the case in which a third party has an SGX equivalent to the case where a data owner has an SGX. In this case, the participants may jointly use the SGX enclave to evaluate the circuit on the data inputs. In order to hide the memory access patterns from the SGX owner, one can implement the ORAM within the enclave. However, this approach is not as efficient as the following fixed-key based garbled circuit technique since it is more efficient to garble a circuit obliviously than to implement the ORAM within an enclave. In a high level description, the data owners and the program owner submit their data/program to the enclave. The enclave garbles the data/program and delivers the garbled circuit/data to the program owner who will evaluate it. Finally, the program owner sends the encrypted garbled circuit output to the enclave which decodes the garbled output, encrypts the output, and sends it to the data owners to decrypt. Note that this case works if any of the data owners have an SGX too, so it is not necessary for the enclave's parent to be a third party.

The initialization phase is almost identical to the one in the first case but with the program owner also sharing a secret key with the enclave using the attestation phase. The online phase, however, differs as the job of this enclave is to garble C_f and not to evaluate it, because evaluating can leak some information about C_f's topology through memory access patterns. When garbling, however, care must be taken as not to leak the functionality of the gates through side channels or the topology of the circuit. To hide the functionality, the garbling scheme must garble each gate in the same way regardless of the gate type; therefore, gate type dependent optimizations such as FreeXOR cannot be used. Instead Garbled Row Reduction techniques can be used together with fixed-key block cipher and point and permute. To hide the circuit topology, gates should be garbled independently from one another. This can be done by generating wire garblings on the fly during garbling on each gate even if they were generated before. Specifically, assume that the circuit C has n gates $G = \{g_1, \ldots, g_n\}$ and l inputs $w_1, \ldots, w_l$. Let $W = \{w_1, \ldots, w_{n+l}\}$ be the collection of wires. Each gate g_i is a tuple $< w_{i1}, w_{i2}, w_{i3}, T_{gi} >$ where w_{i1}, w_{i2} are the input wires, w_{i3} is the output wire, and T_{gi} is the gate type (that is, AND, OR, or NAND). By using the fixed-key garbling technique, the labels $K^0_{w_{i1}}, K^1_{w_{i1}}, K^0_{w_{i2}}, K^1_{w_{i2}}, K^0_{w_{i3}}, K^1_{w_{i3}}$ for the wires w_{i1}, w_{i2}, w_{i3} can be computed on the fly as $K^b_{w_{ij}} = F(R, w^b_{ij})$ for $b = 0, 1$ and $j = 1, 2, 3$, where F is a fixed function and R is a fixed entropy string. Thus given a description of the circuit $C = (W, G)$, one can obtain the garbled circuit by garbling each of the gates $g_1, \ldots, g_n$ in a sequence independently (obliviously). It is noted that for each gate with two incoming wires and one outgoing wire, the garbling scheme needs to compute six labels. For a gate with one incoming wire (e.g., a negation gate) and one outgoing wire, the garbling scheme only needs to compute four labels. Thus, an attacker could use the timing channel to guess whether a gate under garbling process contains two incoming wires or a single incoming wire. To avoid this attack, our above discussion assumes that each gate in circuit C has two incoming wires. In case that the circuit contains gates with a single incoming wire, the garbling scheme should introduce time-delays when garbling such kind of gates.

Accordingly, technique 2 works as follows:

(i) The data owner S_i chooses a random string R_i, encrypts its data input and the R_i using its secret key with the enclave (established during the attestation phase) and sends it to the enclave's parent (third party).

(ii) The circuit owner S_0 chooses a random string R_0, encrypts the circuit and the R_0 using its secret key with the enclave (established during the attestation phase) and sends it to the enclave's parent.

(iii) The enclave's parent (untrusted third party) passes the encrypted data, the encrypted circuit, and the encrypted random strings to the enclave (third party).

(iv) The enclave computes the entropy string $R = H(R_0, R_1, \ldots, R_m)$ for the garbling process where H is a secure hash function.

(v) The enclave garbles the circuit and the input data as explained above.

(vi) The enclave encrypts the garbled circuit and the garbled input data using the secret key shared with the circuit owner (established during the attestation phase) and sends the encrypted values to the circuit owner.

(vii) The circuit owner decrypts the garbled circuit and the garbled input data and evaluates the garbled circuit on the garbled input data.

(viii) The circuit owner encrypts the garbled output using the secret key shared with the enclave and sends it back to the enclave via the enclave's parent.

(ix) The enclave decrypts the garbled output, creates copies of the output for data owners, and encrypts each copy with its appropriate key and sends these encrypted copies to the data owners. Figure 2 explains the technique.

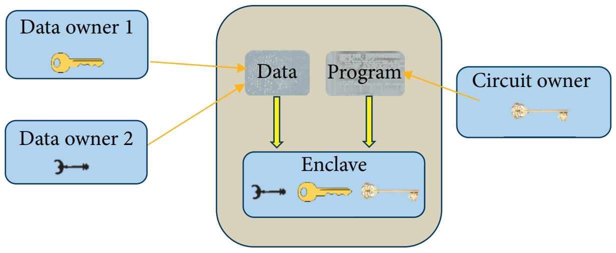

Figure 2: Technique 2. A third party has SGX.

4.3. Encryption. The communication between the data owners and the enclave is encrypted using AES-CBC using the keys that are generated by the attestation process. Initial vectors (IV) are incremented with every encryption in order to eliminate the possibility of having the same cipher when encrypting the same value more than once. The same encryption algorithm is used to secure the communication between the circuit owner and the enclave. Clearly, the enclave should use the appropriate key to decrypt the received data (or circuit) while maintaining a corresponding incrementing for each IV.

5. Security

Our adversary on the enclave's host machine is high-privileged. Accordingly, he can watch all memory access. However, we assume semihonest adversary who has no intention in modifying messages coming from and to the enclave. We assume that the number of parties, the number and size of inputs, and the size of the circuit are known to all parties. We also assume that the inputs, the circuit, and the garbled circuit can all be fit inside the enclave. Enabling paging would allow an enclave to exceed the limitation of 128 MB with an expected slowdown. Despite the fact that the problem of converting a function into a circuit is not a trivial problem, we consider it out of scope since this procedure can be done completely on the program owner before executing the technique.

The security of the two proposed protocols depends on the security of SGX. If the hardware has no faults and performs as intended then SGX should guarantee that the memory of the enclave and the registers used by it as well as the instructions performed by the CPU remain hidden from the host and its operating system in particular. SGX, however, does not protect from side-channel attacks. Hence if something can be observed through measuring the time that certain instructions take, or through the access made to memory or through other indirect means, then the two proposed protocols should protect against that information leakage.

In the first protocol that uses the program owner's SGX, the program stays hidden from the data owners. This is because the program never leaves the program owner's machine and the SGX as created by the program owner himself will not leak information about the program to others. On the other hand, the data is encrypted using an established secret key between the enclave and each data owner using the attestation process. Thus the encrypted data could only be decrypted by the enclave but not by the program owner. By running a directed acyclic circuit, the enclave executes memory access instructions. The time these instructions take and any other observable side-channel issues will look the same regardless of what the input data was. Therefore the data stays hidden from the program owner. After evaluating the circuit the enclave returns the result only to the data owners to prevent the program owner from writing an identity function and learning all the data. Lastly, since the enclave is attested by the data owners they can be sure that the enclave is executing the correct protocol.

In the second protocol, each data owner receives the evaluation output of the garbled circuit on the garbled input data, and the program owner receives the garbled circuit and its garbled input. Since a garbled circuit hides the input throughout computation, the program owner cannot learn anything about the input data. Looking at enclave host, just like in the first protocol all communications between the enclave and the other servers are hidden from the enclave host through encryption. Additionally, the data of the other data owners also remain hidden because they are evaluated in a directed acyclic circuit that produces the same memory accesses and timings regardless of the input data. Since garbling a gate is done in the same way regardless of the gate type, then side channels cannot leak the gate type. In addition, generating the garbling from a permutation (fixed-key block cipher) of the wire labels every time will hide the topology of the circuit and avoid leaking information through memory accesses. Using both a garbling technique that is independent of the gate type and generating wire garblings independently at each gate, ensures that the circuit stays hidden during garbling. Therefore, the program does not leak to any of the data owners and as mentioned before the individual data does not leak to other servers.

6. Experimental Evaluation

In this section, we first explain the setup for our experiments and then we discuss the results of these experiments. We compare the performance of our techniques with FairplayFP, and we analyze the effect of the number of parties (i.e., data owners) on the performance. We also identify the bottleneck step of the second technique that is responsible for the biggest time overhead.

6.1. Experimental Setup. We present two C++ implementations for multiparty private function evaluation using SGX. The source code of our implementations can be downloaded from https://github.com/maanrachid/PFE-SGX:

(i) The first implementation has one circuit owner and an arbitrary number of data owners *np* which is known to all parties. The circuit owner is the parent of the enclave so it creates the enclave, then passes the circuit to it using one ECALL. The data owners send their encrypted data in an arbitrary order to the enclave via its parent. Using a list of *np* different keys, the enclave decrypts the data. The enclave evaluates the circuit and sends encrypted output to the data owners via its parent using one OCALL for each data owner.

(ii) The second implementation has several data owners (*np* data owners), one circuit owner and a server which is the enclave's parent. Both data owners and circuit owner send their data and circuit, respectively, in an arbitrary order to the server. The server passes them to the enclave which decrypts the data and the circuit using a list of *np* +1 keys and performs the garbling for the circuit and the data. Wire's garblings are generated every time they are needed (rather than saving the garbling of a gate's output wire to be used for another gate's input). Two keys were used for garbling: one to generate the 0 garbling and another to generate the 1 garbling. Only point-and-permute is used in this implementation. The enclave sends the garbled circuit and the garbled input back to the circuit owner with all needed keys.

The circuit owner evaluates the garbled circuit on the garbled input and sends the garbled output back to the enclave. The enclave finally decodes the garbled output and sends the output to the data owners.

Our implementations are tested with SGX's SDK version 1.9. The communication between the data owners and the enclave is encrypted using AES-CBC offered by Intel's IPP cryptography library. The same encryption is used for the communication between the enclave and the circuit owner in the second implementation.

Our experiments were run on a machine with 8 GB RAM and Intel(R) Core(TM) i7-6770HQ 2.60 GHz CPU with an enclave's maximum size of 128 MB. The sizes of the input, the output, and the circuit are known to all parties. Data owners, the circuit owner, and the server (enclave's parent) were run on the same machine in all experiments; nevertheless, each program takes the machine name and a port as parameters. Accordingly, they can be run on different machines.

The circuits which we used in our experiments are downloaded from https://homes.esat.kuleuven.be/̃nsmart/MPC/. The descriptions for these circuits are shown in Table 1. The circuits are well-known and they show a diversity in terms of circuit size. We slightly modified the format of each circuit in order to enable an arbitrary number of data owners with arbitrary input shares.

6.2. Results. We compare our implementations with FairplayPF in terms of time consumption. Since FairplayPF has different input format (Secure Function definition Language (SFDL)) than ours, we only run our tests using simple circuits. We study the effect of the number of clients on the performance, and we also analyze the cause of the relative slowdown in our second implementation.

Table 2 shows the elapsed times for one complete round of each technique. These numbers are obtained by running each round 1000 times and recording the average. Time

Table 1: Descriptions of the circuits used in our experiments.

Circuit	# of	# of	Circuit	Garbled
Name	Wires	Gates	Size	Circuit size
Adder32	439	375	6,164	29,692
Multi32	12,438	12,374	198,272	941,740
AES-expanded	29,228	27,692	443,616	2,129,716
AES-nonexpanded	33,872	33,616	538,400	2,559,460
Sha-1	107,113	106,601	1,706,288	8,110,544
DES-expanded	31,233	30,401	486,704	2,324,080
DES-nonexpanded	30,441	30,313	485,296	2,306,128
Md5	78,373	77,861	1,246,320	5,926,176

Circuit size and garbled circuit size are shown in bytes.

Table 2: The elapsed times for one evaluation using both techniques.

Circuit	Technique 1	Technique 2
Adder32	0.0002	0.083
Multi32	0.004	0.1
AES-expanded	0.01	0.173
AES-nonexpanded	0.013	0.195
Sha-1	0.0394	0.528
DES-expanded	0.011	0.181
DES-nonexpanded	0.011	0.169
Md5	0.029	0.37

Time is shown in seconds.

command is applied on one of the data owners in order to obtain the time for each round.

We compared the performance of technique 1 and technique 2 and the relationship with the size of the circuit by sorting the circuits by size (wires and gates). It turns out that with the adder, technique 1 has a speedup 415 times over technique 2, while with the multiplication, technique 1 has a speedup of 23 times. With SHA-1 (the largest), there is a speed up of 13 times which suggests a negative correlation between the size of the circuits and the superiority of technique 1 over technique 2. Figure 3 shows the time consumption for technique 1, technique 2, and FairplayFP. As mentioned before, we restrict our comparisons to these circuits because they are the only circuits which we have in both formats: SFDL (for FairplayFP) and Bristol (for our work). For MD5, for example, An SFDL version of MD5 has to be implemented which is beyond the scope of this work.

We investigate the effect of the number of data owners by running the same experiments using 4 data owners instead of 2 owners. We got the same time consumption for all rounds suggesting that the number of clients has no effect on the performance of any of the techniques.

We studied the major reason behind the slowdown in technique 2. Table 3 shows that garbling is responsible for 60% to 80% of the total time consumption in most of the circuits except for the adder since it is a relatively small circuit and the communication cost has a relatively high cost.

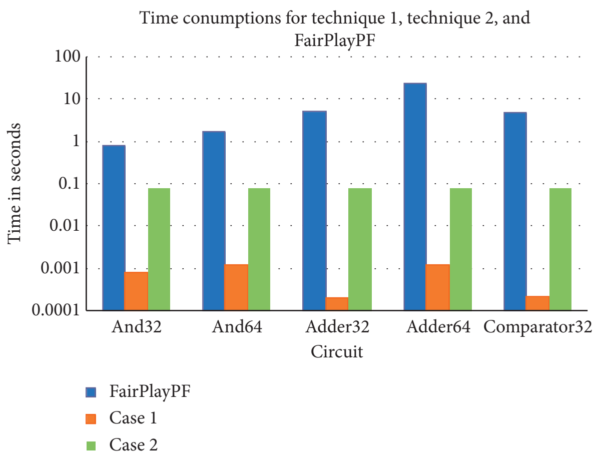

Figure 3: Time consumption for technique 1, technique 2, and FairplayPF.

Table 3: The distribution of time consumption in technique 2.

	Total	Garbling	Reading	Evaluation
Adder32	0.083	0.005	0.0005	0.0006
Multi32	0.1	0.05	0.005	0.005
AES-expanded	0.175	0.11	0.01	0.01
AES-nonexpanded	0.19	0.14	0.01	0.02
Sha-1	0.52	0.42	0.04	0.04
DES-expanded	0.18	0.12	0.014	0.013
DES-nonexpanded	0.17	0.11	0.01	0.01
Md5	0.37	0.29	0.03	0.036

Time is shown in seconds. Reading time is the time required for reading a circuit from a file. Evaluation time is the time required to evaluate the garbled circuit.

7. Conclusion

This work provides a novel method of solving PFE with Intel Software Guard Extension. This method allows for a much more efficient and therefore more practical approach to performing PFE in comparison to previous solutions. The practicality of this solution will open the door to some interesting applications of PFE. Our future work will involve looking at some of these applications and developing a simpler full fledged system that can be used by regular developers to create PFE applications.

Data Availability

The source code of our implementations can be downloaded from https://github.com/maanrachid/PFE-SGX.

Disclosure

The work was mainly performed while the author was working at Qatar University. Contents of the research are solely the responsibility of the authors and do not necessarily represent the official views of the Qatar National Research Fund.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This publication was made possible by a grant from the Qatar National Research Fund (project number NPRP X-063-1-014).

References

[1] V. Kolesnikov and T. Schneider, "A practical universal circuit construction and secure evaluation of private functions," in *Proceedings of the International Conference on Financial Cryptography and Data Security*, pp. 83–97, Cozumel, Mexico, January 2008.

[2] S. Tamrakar, J. Liu, A. Paverd, J.-E. Ekberg, B. Pinkas, and N. Asokan, "The circle game: scalable private membership test using trusted hardware," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 31–44, Abu Dhabi, United Arab Emirates, April 2017.

[3] A. C.-C. Yao, "How to generate and exchange secrets," in *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pp. 162–167, Washington, DC, USA, 1986.

[4] Y. Lindell and B. Pinkas, "A proof of security of Yao' protocol for two-party computation," *Journal of Cryptology*, vol. 22, no. 2, pp. 161–188, 2009.

[5] V. Kolesnikov and T. Schneider, "Improved garbled circuit: free XOR gates and applications," in *Proceedings of the International Colloquium on Automata, Languages, and Programming*, pp. 486–498, Reykjavik, Iceland, July 2008.

[6] 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*, pp. 503–513, Baltimore, MD, USA, May 1990.

[7] V. Kolesnikov, P. Mohassel, and M. Rosulek, "Flexor: flexible garbling for XOR gates that beats free-XOR," in *Proceedings of the International Cryptology Conference on Advances in Cryptology - CRYPTO 2014*, pp. 440–457, Santa Barbara, CA, USA, August 2014.

[8] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole," in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 220–250, Sofia, Bulgaria, April 2015.

[9] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*, pp. 478–492, Berkeley, CA, USA, May 2013.

[10] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams, "Secure two-party computation is practical," in *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*, pp. 250–267, Tokyo, Japan, December 2009.

[11] L. G. Valiant, "Universal circuits (preliminary report)," in *Proceedings of the eighth annual ACM symposium on Theory of computing*, pp. 196–203, Hershey, PA, USA, May 1976.

[12] Á. Kiss and T. Schneider, "Valiant's universal circuit is practical," in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 699–728, Vienna, Austria, May 2016.

[13] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "Tinygarble: highly compressed and scalable sequential garbled circuits," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*, pp. 411–428, San Jose, CA, USA, May 2015.

[14] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella et al., "Fairplay-secure two-party computation system," in *Proceedings of the USENIX Security Symposium*, p. 9, San Diego, CA, USA, August 2004.

[15] J. Katz and L. Malka, "Constant-round private function evaluation with linear complexity," *Lecture Notes in Computer Science*, in *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*, pp. 556–571, Seoul, South Korea, December 2011.

[16] P. Mohassel and S. Sadeghian, "How to hide circuits in MPC an efficient framework for private function evaluation," in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 557–574, Vienna, Austria, May 2013.

[17] P. Mohassel, S. Sadeghian, and N. P. Smart, "Actively secure private function evaluation," in *Proceedings of the Advances in Cryptology–ASIACRYPT 2014*, pp. 486–505, Kaoshiung, Taiwan, December 2014.

[18] S. T. Kent, "Protecting externally supplied software in small computers," Tech. Rep., Massachusetts Institute of Technology Cambridge Laboratory for Computer Science, Cambridge, MA, USA, 1980.

[19] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[20] S. Lu and R. Ostrovsky, "How to garble ram programs?" in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 719–734, Athens, Greece, May 2013.

[21] C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs, "Garbled ram revisited," in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 405–422, Copenhagen, Denmark, May 2014.

[22] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: oblivious memory primitives from intel sgx," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2017.

[23] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi, "HardIDX: practical and secure index with SGX," in *Proceedings of the IFIP Annual Conference on Data and Applications Security and Privacy*, pp. 386–408, Philadelphia,PA, USA, July 2017.

[24] R. Bahmani, M. Barbosa, F. Brasser et al., "Secure multiparty computation from SGX," *Financial Cryptography and Data Security*, in *Proceedings of the International Conference on Financial Cryptography and Data Security*, pp. 477–497, Sliema, Malta, April 2017.

[25] D. Gupta, B. Mood, J. Feigenbaum, K. Butler, and P. Traynor, "Using intel software guard extensions for efficient two-party secure function evaluation," in *Proceedings of the International Conference on Financial Cryptography and Data Security*, pp. 302–318, Christ Church, Barbados, 2016.

[26] K. Nayak, C. Fletcher, L. Ren et al., "Hop: hardware makes obfuscation practical," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium, NDSS*, San Diego, CA, USA, 2017.

[27] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "Iron: functional encryption using intel sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 765–782, Dallas, TX, USA, October 2017.

[28] Y. Yarom and K. Falkner, “Flush+reload: a high resolution, low noise, l3 cache side-channel attack,” in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association, San Diego, CA, USA, pp. 719–732, August 2014.
[29] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-tenant side-channel attacks in paas clouds,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS ’14*, pp. 990–1003, New York, NY, USA, November 2014.
[30] D. Gullasch, E. Bangerter, and S. Krenn, “Cache games – bringing access-based cache attacks on aes to practice,” in *Proceedings of the 2011 IEEE Symposium on Security and Privacy, ser. SP ’11*, IEEE Computer Society, Washington, DC, USA, pp. 490–505, November 2011.
[31] J. van de Pol, N. P. Smart, and Y. Yarom, “Just a little bit more,,” in *Proceedings of the Cryptographers’ Track at the RSA Conference*, San Francisco, CA, USA, February 2015.
[32] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in *Proceedings of the 2015 IEEE Symposium on Security and Privacy, ser. SP ’15*, IEEE Computer Society, Washington, DC, USA, pp. 605–622, May 2015.