Actively Secure Private Function Evaluation

Payman Mohassel[1,2], Saeed Sadeghian[1], and Nigel P. Smart[3]

[1] Dept. Computer Science, University of Calgary,
pmohasse@ucalgary.ca, sadeghis@ucalgary.ca
[2] Yahoo Labs,
pmohassel@yahoo-inc.com
[3] Dept. Computer Science, University of Bristol,
nigel@cs.bris.ac.uk

Abstract. We propose the first general framework for designing actively secure private function evaluation (PFE), not based on universal circuits. Our framework is naturally divided into pre-processing and online stages and can be instantiated using any generic actively secure multiparty computation (MPC) protocol.

Our framework helps address the main open questions about efficiency of actively secure PFE. On the theoretical side, our framework yields the first actively secure PFE with linear complexity in the circuit size. On the practical side, we obtain the first actively secure PFE for arithmetic circuits with $O(g \cdot \log g)$ complexity where g is the circuit size. The best previous construction (of practical interest) is based on an arithmetic universal circuit and has complexity $O(g^5)$.

We also introduce the first linear Zero-Knowledge proof of correctness of "extended permutation" of ciphertexts (a generalization of ZK proof of correct shuffles) which maybe of independent interest.

Keywords. Secure Multi-Party Computation, Private Function Evaluation, Malicious Adversary, Zero-Knowledge Proof of Shuffle

1 Introduction

Private Function Evaluation (PFE) is a special case of Multi-Party Computation (MPC), where the parties compute a function which is a private input of one of the parties, say party P_1. The key additional security requirement is that all that should leak about the function to an adversary, who does not control P_1, is the size of the circuit (i.e. the number of gates and distinct wires within the circuit). Clearly, PFE follows immediately from MPC by designing an MPC functionality which implements a universal machine/circuit; thus the only open questions in PFE research are those of efficiency. Using universal circuits one can achieve complexity of $O(g^5)$ in case of arithmetic circuits [23] and $O(g \cdot \log g)$ for boolean circuits [26]. For ease of exposition we ignore the factors depending on the number of parties and the security parameters as they depend on the particular underlying MPC being used. We still provide some numbers for the specific SPDZ instantiation in section 5.

A number of previous work [1,2,4,12,14,15,16,17,22,24] have considered the design and implementation of more efficient general- and special-purpose private function evaluation. A major motivation behind these solutions (and PFE in general) is to hide the function being computed since it is proprietary, private or contains sensitive information. Some applications of interest considered in the literature are software diagnostic [4], medical applications [2], and intrusion detection systems [20].

But all prior solutions are in the semi-honest model and fail in the presence of an active adversary who does not follow the steps of the protocol (with the exception of the generic approach of applying an actively secure MPC to universal circuits). For example, a malicious party who does not own the function can cheat to learn the proprietary function or modify the outcome of computation without the function-holders' knowledge. Or a malicious function-holder, can learn information about honest parties' inputs.

One may question the need for actively secure PFE as the function-holder can cheat and use a malicious function, which reveals information about the other party's input. While we consider the general scenario in our protocols, there are common practical scenarios where the function-holder has no output in the computation, and therefore maliciously changing the function still does not let him learn anything even if he is actively cheating.

1.1 Our Contribution

In this work, we present the first general framework for designing actively secure PFE, not based on universal circuits. Our framework can be instantiated upon a generic actively secure MPC protocol satisfying quite general properties; namely that they are secret sharing based, actively secure (either robust or with aborts), can implement reactive functionalities, and have an ability to open various sharings securely, as well as generate (efficiently) sharings of random values. Suitable actively secure MPC protocols include BDOZ [3] and SPDZ [8] (for the case of arithmetic circuits and an arbitrary number of players with a dishonest majority), Tiny-OT [19] (for binary circuits and two players), or protocols such as that implemented in VIFF [7] utilizing Shamir secret sharing with a threshold of $t < n/3$.

Our framework helps address the main open questions about efficiency of actively secure PFE. On a theoretical note, we use it to show that actively secure PFE with linear complexity (in circuit size) is indeed feasible while avoiding strong primitives such as fully-homomorphic encryption (FHE).[4] On a practical

[4] Note that with the use of the right circuit-private FHE scheme [21], and appropriate ZK proofs for correctness of the computation on encrypted data, it is likely possible to achieve linear PFE based on FHE, but we are interested in the use of much weaker primitives such as singly homomorphic encryption.

note, we obtain a practical actively secure PFE for arithmetic circuit with $O(g \cdot \log g)$ complexity (a significant reduction from $O(g^5)$ [23]), and the first actively secure PFE in the information-theoretic setting.

Our Framework. Our framework can be seen as an extension of the new framework of [17] which is only secure against passive adversaries. The key idea in [17] is to divide the problem into two sub-problems, the problem of hiding the topology of the wiring between individual gates (topology hiding), and the problem of hiding exactly what gate is evaluated (gate hiding), i.e. an addition or a multiplication (or AND/OR/XOR in case of boolean circuits).

This framework yields better asymptotic and practical efficiency for passively secure PFE compared to the universal circuit approach (see [17] for a detailed efficiency comparison). An important open question is then how to extend their solution to the case of active adversaries efficiently. In this paper we do exactly that by providing a recipe for turning any actively secure MPC protocol that satisfies our general requirements into an actively secure PFE protocol.

Our framework operates in two phases, an offline phase and an online phase. As in the case of standard MPC in the pre-processing model, our offline phase is input independent but it depends on the function. The offline phase is use-once, in the sense that the data produced cannot be reused for multiple invocations of the online phase. We note that a similar function-dependent pre-processing model (referred to as *dedicated pre-processing*) was recently considered in [9]. Dedicated pre-processing is particularly natural in PFE applications where the sensitive/proprietary function stays fixed for a period of time and is used in multiple executions (clearly in the latter case we need to execute the pre-processing multiple times, but this can be done in advance). Of course, if one is not willing to count a function-dependent offline phase as valid, then our complexities would be the combination of the two phases. It maybe the case that our underlying MPC protocol is itself in the pre-processing model (e.g. [3,8,19]), in which case that pre-processing will be essentially independent of the input and function being evaluated. Our framework shows the feasibility of offline computation independent of inputs, which was not the case in [17]. We elaborate on the two phases next:

Offline Phase. Roughly speaking, our offline phase generates two vectors of random values, *maps* the second to a new vector using a mapping that captures the topology of the circuit (referred to as extended permutation in [17]), and subtracts the result from the first. The result of the subtraction (difference vector) is opened while the two original vectors are shared among the parties. The two random vectors are used as one-time pads of all the intermediate values in the circuit, while the "difference vector" is used by the function-holder to connect the output of one gate to the input of another without learning the values or revealing the circuit topology. The offline phase also generates one-time MACs of all the components of the "difference vector" computed above, using a fixed global MAC key. These MACs are used to check the function-holder's work in the online phase of the protocol. These steps commit P_1 privately to the topology of the circuit. We also privately commit P_1 to gate types, hence fully committing him to the function being computed.

Online Phase. Our online, or circuit evaluation, phase is very distinct from that deployed in the underlying MPC protocol we use. In existing instantiations of our underlying MPC protocol, parties evaluate gates on values whose secrecy is maintained due to the fact that one is working on secret shared values only. In our protocol the parties have public one-time pad encryptions of the values being computed on, but the encryption keys, which are the random values generated in the offline phase, remain secret-shared. Party P_1 (the function holder) then uses the random vectors computed in the offline phase to transform the encrypted output of one gate to the encrypted input of the upcoming gate while maintaining one-time MACs of all the values he computes. These MACs allow all other parties to check P_1's work without learning the circuit topology. These operations are carried out securely using the underlying MPC protocol.

In both the online and the offline phase, all parties check P_1's work by checking the MACs of the values he computes locally. If any of the MACs fail, in case of security with abort, parties can simply end the protocol. But in case of robust MPC (e.g. $t < n/3$ for robust information theoretically secure protocols) the protocol needs to continue without P_1. To achieve this, honest parties jointly recover P_1's function and play his role in the remainder of the protocol.

In our protocols, if any adversary deviates from the protocol then, except with negligible probability, the honest parties will either abort, or be able to recover from the introduced error. The exact response depends on the underlying MPC protocol on which our PFE protocol is built. In all cases the privacy of the honest players inputs is preserved, bar what can be obtained from the output of the private function chosen by player P_1. Note that P_1 may or may not be a recipient of output, but many application of PFE are concerned with scenarios where the function-holder has no output.

Efficient Instantiations. One can efficiently instantiate our online phase with a linear complexity, using any actively secure MPC satisfying our requirements. The main challenge, therefore, lies in efficient instantiation of the offline phase. It is possible to implement our offline phase using any actively secure MPC sub-protocol as well (by securely computing a circuit that performs the above mentioned task) but the resulting constructions would neither be linear nor constant-round.

- We introduce a instantiation with $O(g)$ complexity, proving the feasibility of linear actively secure PFE for the first time. Our main new technical ingredient is a linear zero-knowledge (ZK) proof of "correct extended permutation" of ElGamal ciphertexts. While linear ZK proofs of shuffles are well-studied, it is not clear how to extend the techniques to extended permutation (see our incomplete attempt in the full version [18]) Instead, we propose a generic and linear solution that uses ZK proof of a correct shuffle in a black-box manner, and may be of independent interest. Our solution is based on the switching network construction of EP [17]. This construction consists of three components, two of which are permutation networks. Instead of evaluating switches, we use singly homomorphic encryption to evaluate each component, and then re-randomize. We use existing ZK proofs of shuffle to prove the correctness of first and third components which perform permutation. The middle component requires a separate compilation of ZK protocols. Note that generically applying ZK proofs to UC circuit evaluation does not provide a linear solution, and applying ZK proofs for the EP component also does not work. Our customized linear $\mathcal{ZK}_{\mathrm{EP}}$ gets around these problems.

- We introduce a *constant-round* instantiation with $O(g \cdot \log g)$ complexity (contrast with $O(g^5)$ complexity for universal arithmetic circuits) that is also of practical interest. Our technique is itself an extension of ideas from [17]. In particular the basic algorithm is that of [17] for oblivious evaluation of a switching network, but some care needs to be taken to make sure the protocol is actively secure. This is done by applying MACs to the data being computed on. However, instead of having the MAC values being secret shared (as in SPDZ) or kept secret (as in BDOZ and Tiny-OT), the MAC values are public with the keys remaining secret shared. Nevertheless, the MACs used are very similar to those used in the BDOZ and Tiny-OT protocols [3,19], since they are two-key MACs in which one key is a per message key and one is a global key. While using MAC's is quite standard for ensuring consistency of data, our efficient deployment in the framework is non-trivial and novel. For example, while addition of MACs in the offline phase is done using a generic MPC, the circuit evaluation (online phase) does not use an MPC. This is different from [17]'s approach and previous MPC work. General active security techniques can not be directly employed in this context. It is not clear how to use cut-and-choose in case of PFE, e.g. it is not clear how not to reveal the function in the opening, and there are additional components (i.e. EP) in a PFE protocol which cut-and-choose does not seem to resolve.

Efficiency Discussion. We emphasize that our linear complexity solution is a feasibility result at it was an open question whether active PFE with linear complexity in circuit size is possible given simple crypto primitive such as singly homomorphic encryption (as opposed FHE). Our "efficient" arithmetic PFE only requires $O(g \log g)$ multiplication gates and it is a significant improvement in comparison with applying of arithmetic MPC to universal arithmetic circuit of size $O(g^5)$ [23]. If we apply active secure MPC for arithmetic circuits to this universal circuit the complexity cannot get better than $O(g^5)$. One can turn an arithmetic circuit into a boolean circuit and use Valiant's boolean UC [26] to obtain a PFE. But this is highly inefficient, and therefore we do not discuss this in detail.

2 Notation and the Underlying MPC Protocol

We assume our function f to be evaluated will eventually be given by player P_1 as an arithmetic circuit over a finite field $\mathbf{F}_p$; note p may not necessarily be prime. We let $\mathbf{g}(f)$ denote the number of gates in the circuit representing f. For gates with fan-out greater than one, we count each seperate output wire as a different wire. We also select a value k such that $p^k > 2^{\mathsf{sec}}$, where sec is the security parameter; this is to ensure security of our MAC checking procedure in the online phase.

We assume n parties $P_1, \ldots, P_n$, of which an adversary may corrupt (statically) up to t of them; the value of t being dependent on the specific underlying MPC protocol. The corrupted adversaries could include party P_1. The MPC protocol should implement the functionality described in Figure 1. This functionality is slightly different from standard MPC functionalities in that we try to capture both the honest majority and the dishonest majority setting; and in the latter setting the adversary can force the functionality to abort at any stage of the computation and not just the output. We also introduce another operation called **Cheat** which will be useful in what follows.

It is clear that modern actively secure MPC protocols such as [7,8,19], implement this functionality in different settings. Thus various different settings (i.e. different values of n, p and t) will be able to be dealt with in our resulting PFE protocol by simply plugging in a different underlying MPC protocol. To ease exposition later we express our MPC protocol as evaluating functions in the finite field $\mathbf{F}_{p^k}$. Clearly such an MPC protocol can be built out of one which evaluates functions over the base finite field $\mathbf{F}_p$.

To ease notation in what follows we shall let $[\mathit{varid}]$ denote the value stored by the functionality under (varid, a); and will write $[z] = [x] + [y]$ as a shorthand for calling **Add** and $[z] = [x] \cdot [y]$ as a shorthand for calling **Multiply**. And by abuse of notation we will let varid denote the value, x, of the data item held in location (varid, x).

3 Our Active PFE Framework

In this section we describe our active PFE framework in detail. We start by describing the offline functionality which pre-processes the function/circuit the parties want to compute (Section 3.1). Then, in Section 3.2, we show that given a secure implementation of $\mathcal{F}_{\text{OFFLINE}}$, one can efficiently (linear complexity) construct an actively secure PFE based on any actively secure MPC. We postpone efficient instantiations of $\mathcal{F}_{\text{OFFLINE}}$ to later sections.

3.1 The Function Pre-Processing (Offline) Phase

In this section we detail the requirements of our pre-processing step once player P_1 has decided on the function f to be evaluated. P_1 is only required to enter a valid circuit, equivalent to his function f into the protocol. Each non-output

Functionality $\mathcal{F}_{\text{MPC}}$

The functionality consists of seven externally exposed commands **Initialize**, **Cheat**, **Input Data**, **Random**, **Add**, **Multiply**, and **Output** and one internal subroutine **Wait**.

Initialize: On input $(init, p, k, flag)$ from all parties, the functionality activates and stores p and k; and a representation of $\mathbf{F}_{p^k}$. The value of *flag* is assigned to the variable dhm, to signal whether the MPC functionality should operate in the dishonest majority setting. The set of "valid" players is initially set to all players. In what follows we denote the set of adversarial players by $\mathcal{A}$.

Cheat: This is a command which takes as input a player index i, it models the case of (most) robust MPC protocols in the honest majority case. On execution the functionality aborts if dhm is set to *true*. Otherwise the functionality waits for input from all players. If a majority of the players return *OK* then the functionality reveals all inputs made by player i, and player i is removed from the list of "valid" players (the functionality continues as if player i does not exist).

Wait: This does two things depending on the value of dhm.

- If dhm is set to *true* then it waits on the environment to return a *GO/NO-GO* decision. If the environment returns *NO-GO* then the functionality aborts.
- If dhm is set to *false* then it waits on the environment. The environment will either return *GO*, in which case it does nothing, or the environment returns a value $i \in \mathcal{A}$, in which case $\mathsf{Cheat}(i)$ is called.

Input Data: On input $(input, P_i, varid, x)$ from P_i and $(input, P_i, varid, ?)$ from all other parties, with *varid* a fresh identifier, the functionality stores $(varid, x)$. The functionality then calls **Wait**.

Random: On command $(random, varid)$ from all parties, with *varid* a fresh identifier, the functionality selects a random value r in $\mathbf{F}_{p^k}$ and stores $(varid, r)$. The functionality then calls **Wait**.

Add: On command $(add, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x + y)$. The functionality then calls **Wait**.

Multiply: On input $(multiply, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x \cdot y)$. The functionality then calls **Wait**.

Output: On input $(output, varid)$ from all honest parties (if *varid* is present in memory), the functionality retrieves $(varid, x)$ and outputs it to the environment. The functionality then calls **Wait**, and only if **Wait** does not abort then it outputs x to all players.

Fig. 1: The required ideal functionality for MPC

wire w in the circuit is connected at one end (which we shall call the *outgoing wire or left point*) to a source, this is either the output of a (non-output) gate or an input wire. Conversely each non-output wire is connected at the other end (which we shall call the *incoming wire or right point*) to a destination point which is always an input to a gate. We denote the number of distinct Incoming Wires on the right by $\mathsf{iw}(f)$. We let $\mathsf{ow}(f)$ denote the number of Outgoing Wires on the left. Note that $\mathsf{iw}(f) = 2g$ and $\mathsf{ow}(f) = n+g-o$ where o is the number of output gates in the circuit. Since we are dealing with arbitrary fan out we have that $\mathsf{ow}(f) \leq \mathsf{iw}(f)$.

Functionality $\mathcal{F}_{\textsc{Offline}}$

Initialize: As for $\mathcal{F}_{\text{MPC}}$.
Wait: As for $\mathcal{F}_{\text{MPC}}$.
Input Data: As for $\mathcal{F}_{\text{MPC}}$.
Cheat: As for $\mathcal{F}_{\text{MPC}}$.
Random: As for $\mathcal{F}_{\text{MPC}}$.
Add: As for $\mathcal{F}_{\text{MPC}}$.
Multiply: As for $\mathcal{F}_{\text{MPC}}$.
Output: As for $\mathcal{F}_{\text{MPC}}$.
Input Function: On input $(\mathit{inputfunction}, \pi, f)$ from player P_1 the functionality performs the following operations

- The functionality calls (random, K).
- If f is not a valid arithmetic circuit then the functionality aborts.
- For $i \in \{1, \ldots, \mathsf{iw}(f)\}$ the functionality calls (random, r_i) and (random, s_i).
- For $j \in \{1, \ldots, \mathsf{ow}(f)\}$ the functionality calls (random, l_j) and (random, t_j).
- The functionality then computes, for all $i \in \{1, \ldots, \mathsf{iw}(f)\}$

$$[p_i] = [r_i] - [\ell_{\pi(i)}], \quad [q_i] = ([s_i] - [t_{\pi(i)}]) + ([r_i] - [\ell_{\pi(i)}]) \cdot [K]$$

- The functionality then outputs (p_i, q_i) to all players, for $i \in \{1, \ldots, \mathsf{iw}(f)\}$, by calling (output, p_i) and (output, q_i).
- For $i \in \{1, \ldots, g\}$ the functionality calls $(\mathit{input}, P_1, G_i, 0)$ if gate i in the description of f is an addition gate, and $(\mathit{input}, P_1, G_i, 1)$ if gate i is a multiplication gate.

Fig. 2: The required ideal functionality for the Offline Phase

To fully capture the topology of the circuit we give each outgoing wire and incoming wire in the circuit a unique label. The labels for the outgoing wires will be $\{1, \ldots, \mathsf{ow}(f)\}$ starting from the input wires and then moving to the output wires of each gate in a topological order decided by P_1, whilst the labels for the incoming wires will be $\{1, \ldots, \mathsf{iw}(f)\}$ labelling the input wires to each gate in the same topological order. The topology is then defined by a mapping from outgoing wires to incoming wires and is called an "extended permutation" in [17]. We denote the inverse of this mapping by a function π from $\{1, \ldots, \mathsf{iw}(f)\}$

onto $\{1, \ldots, \mathsf{ow}(f)\}$. If w is a wire in the circuit with incoming wire label i, then it's outgoing wire label is given by $j = \pi(i)$.

To execute the function pre-processing, player P_1 on input of f determines a mapping π corresponding to f. The offline phase functionality $\mathcal{F}_{\textsc{Offline}}$ which is described in Figure 2, extends the $\mathcal{F}_{\text{MPC}}$ functionality of Figure 1 by adding an additional operation **Input Function**. The **Input Function** generates a vector of random (but correlated) values and their one-time MACs using a fixed global MAC key K. In particular, the functionality first stores a vector of random values (r_i) for each incoming wire and another vector of random values (ℓ_i) for the outgoing wires in the circuit. These random values will play the role of "pads" for one-time encryption of the computed wire values in the online phase. The functionality then computes p_i, the difference between each outgoing wire's value r_i and the corresponding incoming wires' value $\ell_{\pi(i)}$, and reveals p_i to all parties. This difference vector will allow P_1 to maintain one-time encryption of each wire value in the online phase without revealing the circuit topology. Additional random values (s_i, t_i) and the global MAC key K are used to compute one-time MACs of each p_i, namely q_i. These MACs will be used to check P_1's actions in the online phase. The **Input Function** also commits P_1 to the function of each gate in his circuit by storing a bit (0 for addition and 1 for multiplication) for each gate.

3.2 The Function Evaluation (Online) Phase

We can now present our framework for actively secure PFE. We wish to implement the functionality in Figure 3. We express the functionality as evaluating a function f provided by P_1 which takes as input n inputs in $\mathbf{F}_{p^k}$, one from each player. Again we present the functionality in both the honest majority and the dishonest majority settings.

Realizing $\mathcal{F}_{\textbf{Online}}$ Given $\mathcal{F}_{\textbf{Offline}}$ and $\mathcal{F}_{\textbf{MPC}}$ A generic instantiation of $\mathcal{F}_{\textsc{Offline}}$ based on any MPC is give in Figure 5. The idea is to work with *one-time pad* encryptions of the values for all intermediate wires and the corresponding one-time MACs. Here, the pads (r, ℓ, s, t values), as well as the MAC Key K are generated by the offline functionality, and shared among the parties so no party can learn intermediate values or forge MACs on his own.

In more detail, the protocol proceeds as follows. Initially, parties compute one-time encryption of the input values to the circuit (pads are the corresponding ℓ values). Then, the following process is repeated for every gate in the circuit until every gate is processed. Parties then open the outcome of the output gates as their final result.

For each gate, party P_1 uses the "difference vectors" (p_i values) from the offline phase to transform the one-time encryption of output of the previous gate to the one-time encryption of input of the current gate (the result is denoted by d_{i_0}, d_{i_1} for the i-th gate.), without revealing the topology or learning the actual wire values. This is diagrammatically presented in Figure 4 to aid the reader. A

Functionality $\mathcal{F}_{\text{Online}}$

Initialize: On input $(\mathit{init}, p, k, \mathit{flag})$ from all players, the functionality activates and stores p and k; and a representation of $\mathbf{F}_{p^k}$. The value of *flag* is assigned to the variable dhm, to signal whether the underlying MPC functionality should operate in the dishonest majority setting.

Wait: If dhm is set to *false* then this does nothing. Otherwise it waits on the environment to return a *GO*/*NO-GO* decision. If the environment returns *NO-GO* then the functionality aborts.

Input Function: On input $(\mathit{inputfunction}, f)$ from player P_1 the functionality stores $(\mathit{function}, f)$. The functionality now calls **Wait**.

Input Data: On input $(\mathit{input}, P_i, x_i)$ from player P_i the functionality stores (input, i, x_i). The functionality now calls **Wait**.

Output: On input (output) from all honest players the functionality retrieves the data x_i stored in (input, i, x_i) for $i \in \{1, \ldots, n\}$ (if all do not exist then the functionality aborts). The functionality then retrieves f from $(\mathit{function}, f)$ and computes $y = f(x_1, \ldots, x_n)$ and outputs it to the environment (or aborts if $(\mathit{function}, f)$ has not been stored). The functionality now calls **Wait**. Only on a successful return from **Wait** will the functionality output y to all players.

Fig. 3: The required ideal functionality for PFE

similar transformation is done on MACs of the wire values (using q_i values) in order to keep P_1 honest in his computation (denoted by m_{i_0}, m_{i_1}).

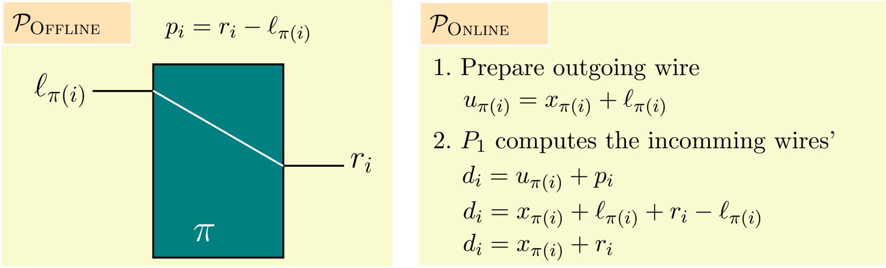
$\mathcal{P}_{\text{Offline}}$
$p_i = r_i - \ell_{\pi(i)}$
$\ell_{\pi(i)}$
r_i
π
$\mathcal{P}_{\text{Online}}$
1. Prepare outgoing wire
$u_{\pi(i)} = x_{\pi(i)} + \ell_{\pi(i)}$
2. P_1 computes the incomming wires'
$d_i = u_{\pi(i)} + p_i$
$d_i = x_{\pi(i)} + \ell_{\pi(i)} + r_i - \ell_{\pi(i)}$
$d_i = x_{\pi(i)} + r_i$

Fig. 4: Transformation of one-time encryption of an outgoing wire to the one-time encryption of an incoming wire using the values computes in $\mathcal{P}_{\text{Offline}}$ protocol.

Then, the protocol proceeds by jointly removing the one-time pads for the two inputs of the current gate and evaluating it together in order to compute a shared output z_i. Note that in this gate evaluation the gate type G_i is secret and shared among the players. This step can be performed using the $\mathcal{F}_{\text{MPC}}$ operations. Then, parties compute a one-time encryption of z_i using the corresponding ℓ value as the pad, and denote the result by u_j, just a relabeling where j is the outgoing wire's label of the output wire of the gate (note that $j = n + i$ since the outgoing wires are labeled starting with the n input wires and then the output wire of each gate).

Note, that if P_1 tries to deviate from the protocol in his local computation (i.e. when he connects outgoing wires to incoming wires) the generated MACs will not pass the jointly performed verifications and he will be caught. In that case, either the protocol aborts (in the case of dishonest majority) or his input (i.e. the function) is revealed (in the case of honest majority).

This leads to the following theorem, whose proof is given in full version [18].

Theorem 1. *In the $\mathcal{F}_{\text{OFFLINE}}$-hybrid model the protocol in Figure 5 securely implements the PFE functionality in Figure 3, with complexity $O(g)$.*

4 Implementing $\mathcal{F}_{\text{Offline}}$ with Linear Complexity

In this section we give a linear instantiation of the offline phase of the framework. Since our online phase has linear complexity, a linear offline phase implementation leads to a linear actively secure PFE. The main challenge in obtaining a linear solution is to design a linear method for applying the extended permutation π to values $\{[\ell_i]\}$ and $\{[t_i]\}$ to produce shared values $\{[\ell_{\pi(i)}]\}$ and $\{[t_{\pi(i)}]\}$. In the semi-honest case [17], linear complexity solution for this problem is achieved by employing a singly homomorphic encryption. The shared values are jointly encrypted; P_1 applies the extended permutation to the resulting ciphertexts and re-randomizes them in order to hide π; parties jointly decrypt in order to obtain the shares of the resulting plaintexts. To obtain active security, we need to make each step of the following computation actively secure:

1. Players encrypt the shared input (all of which lie in $\mathbf{F}_{p^k}$) using an encryption scheme, with respect to a public key for which the players can execute a distributed decryption protocol. The resulting ciphertexts are sent to P_1.
2. Player P_1 applies the EP and re-randomizes the ciphertexts and sends them back. He then uses the $\mathcal{ZK}_{\text{EP}}$ protocol to prove his operation has been done correctly.
3. The players then decrypt the permuted ciphertexts and recover shares of the plaintexts.

To implement the first and last steps we use an an instantiation based on ElGamal encryption, see full version [18]. The middle step is more tricky, and we devote the rest of this section to describing this. For the middle step we need a linear zero-knowledge protocol to prove that P_1 applied a valid EP to the ciphertexts. Proof of a correct shuffle is a well studied problem in the context of Mix-Nets, and linear solutions for it exist [11]. As discussed in full version[18] , however, extending these linear proofs to the case of extended permutations faces some subtle difficulties which we leave as an open question. Instead we aim for a more general construction that uses the currently available proofs of shuffling, in a black-box way.

Protocol $\mathcal{P}_{\text{Online}}$

The protocol is described in the $\mathcal{F}_{\text{Offline}}$-hybrid model.

Input Function: Player P_1 given f selects the switching network mapping π and then calls $(\mathit{inputfunction}, \pi, f)$ on the functionality $\mathcal{F}_{\text{Offline}}$.

Input Data: On input $(\mathit{input}, P_i, x_i)$ from player P_i the protocol executes the (input, i, x_i) operation of the functionality $\mathcal{F}_{\text{Offline}}$.

Output: The evaluation of the function proceeds as follows; where for ease of exposition we set $x_{\pi(h)} = y_h$ for all h, i.e. if a wire has input x_i on the left (as outgoing wire) then it has the same value y_h on the right (as incoming wire) where $i = \pi(h)$

- **Preparing Inputs to the Circuit:**
  - For each input wire i $(1 \leq i \leq n)$ the players execute $[u_i] = [x_i] + [\ell_i]$, where i is the outgoing wire's label corresponding to that input wire, and $[v_i] = [t_i] + ([x_i] + [\ell_i]) \cdot [K]$ using the $\mathcal{F}_{\text{MPC}}$ functionality available via $\mathcal{F}_{\text{Offline}}$.
  - Parties then call (output, u_i) and (output, v_i) to open $[u_i]$ and $[v_i]$.
- **Evaluating the Circuit:** For every gate $1 \leq i \leq g$ in the circuit players execute the following (here we assume that the gates are indexed in the same topological order P_1 chose to determine π):
  - P_1 **Prepares the Two Inputs for Gate** $\boldsymbol{i}$**.**
    - ∗ Note that the two input wires for gate i have incoming wire labels $i_0 = 2i - 1$ and $i_1 = 2i$, and the (u, v) value for their corresponding outgoing wire labels are already determined, i.e. $u_{\pi(i_j)}$ and $v_{\pi(i_j)}$ are already opened for $j \in \{0, 1\}$.
    - ∗ Player P_1 computes, for $j = 0, 1$,

$$\begin{aligned} d_{i_j} = u_{\pi(i_j)} + p_{i_j} &\doteq (y_{i_j} + \ell_{\pi(i_j)}) + (r_{i_j} - \ell_{\pi(i_j)}) \\ &\doteq y_{i_j} + r_{i_j}, \\ m_{i_j} = v_{\pi(i_j)} + q_{i_j} &\doteq (t_{\pi(i_j)} + (y_{i_j} + \ell_{\pi(i_j)}) \cdot K) \\ &\quad + \big((s_{i_j} - t_{\pi(i_j)}) + (r_{i_j} - \ell_{\pi(i_j))}) \cdot K\big) \\ &\doteq s_{i_j} + (y_{i_j} + r_{i_j}) \cdot K. \end{aligned}$$

    - ∗ Player P_1 then broadcasts the values d_{i_j} and m_{i_j} to all players.
  - **Players Check** P_1**'s Input Preparation.**
    - ∗ All players then use the $\mathcal{F}_{\text{MPC}}$ operations available (via the interface to the $\mathcal{F}_{\text{Offline}}$ functionality) so as to store in the $\mathcal{F}_{\text{MPC}}$ functionality the values $[n_{i_j}] = [s_{i_j}] + (y_{i_j} + r_{i_j}) \cdot [K]$. The value is then opened to all players by calling $(\mathit{Output}, n_{i_j})$.
    - ∗ If $n_{i_j} \neq m_{i_j}$ then the players call **Cheat**(1) on the $\mathcal{F}_{\text{MPC}}$ functionality. This will either abort, or return the input of P_1 (and hence the function), in the latter case the players can now proceed with evaluating the function using standard MPC and without the need for P_1 to be involved.
  - **Players Jointly Evaluate Gate** $\boldsymbol{i}$**.**
    - ∗ The players store the value $[y_{i_j}] = d_{i_j} - [r_{i_j}]$ in the $\mathcal{F}_{\text{MPC}}$ functionality.
    - ∗ The $\mathcal{F}_{\text{MPC}}$ functionality is then executed so as to compute the output of the gate as

$$[z_i] = (1 - [G_i]) \cdot ([y_{i_0}] + [y_{i_1}]) + [G_i] \cdot [y_{i_0}] \cdot [y_{i_1}].$$

    - ∗ Note that the outgoing wire label corresponding to the output wire of the ith gate is $j = n + i$ so we just relabel $[z_i]$ to $[z_j]$.
    - ∗ If G_i is an output gate, players call (Output, z_i) to obtain z_i, disregard next steps and continue to evaluate next gate.
    - ∗ The players compute via the MPC functionality $[u_j] = [z_j] + [\ell_j]$.
    - ∗ The players call (Output, u_j) so as to obtain u_j.
    - ∗ The players then compute via the MPC functionality

$$[v_j] = [t_j] + u_j \cdot [K] \doteq [t_j + (z_j + \ell_j) \cdot K].$$

    - ∗ The players call (Output, v_j) so as to obtain v_j.

Fig. 5: The Protocol for implementing PFE

4.1 Linear $\mathcal{ZK}_{\mathbf{EP}}$ Protocol

After players compute the encryption of the shared inputs, P_1 knowing the circuit topology, applies the corresponding extended permutation to the ciphertexts. He then re-randomizes the ciphertexts and then "opens" the ciphertexts. Next, we give a linear zero-knowledge protocol $\mathcal{ZK}_{\mathrm{EP}}$, which enables P_1 to prove the correctness of his operation (i.e final ciphertexts are the result of P_1 applying a valid EP to the input ciphertexts). As our first attempt we considered the possibility of extending existing linear proofs of shuffle to get linear proofs of extended permutation. While plausible there are subtle difficulties that need to be addressed. For more details regarding our attempt on extending the method of Furukawa [11,10], refer to full version[18]. We leave this approach as an open problem. Instead we give a more general construction which makes black-box calls to proof of shuffle. This construction is inspired by the switching network construction of EP given in [17]. We first revisit the extended permutation construction of [17].

Assume the EP mapping represented by the function: $\pi : \{1...n\} \rightarrow \{1...m\}$ (Which maps m input wires to n output wires ($n \geq m$)). Note that in this section we use n and m to denote the size of EP. In a switching network, the number of inputs and outputs are the same, therefore, the construction takes m real inputs of the EP and $n - m$ additional *dummy inputs*. The construction is divided into three components. Each component takes the output of the previous one as input. Instead of applying the EP in one step, P_1 applies each component separately and uses a zero-knowledge protocol to prove its correctness. Figure 6 demonstrates the components. Next, we describe each component and identify the required ZK proof.

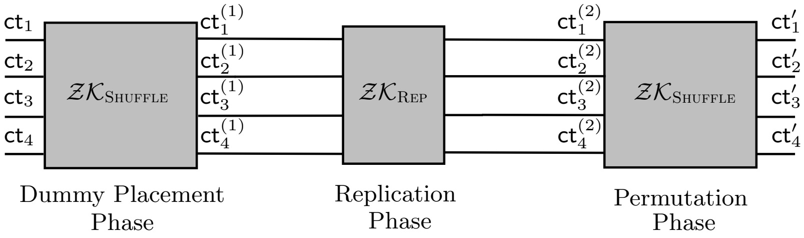

Fig. 6: EP construction. Components' names are written underneath. The zero-knowledge protocol for each component is written inside it's component box.

Table 1 lists the zero-knowledge protocols that we make a black-box use in our $\mathcal{ZK}_{\mathrm{EP}}$ protocol. Note that we use P and Q for our EC instantiation instead of g and h.

- **Dummy-value placement component:** This takes the real and dummy ciphertexts as input and for each ciphertexts of a real value that is mapped

$\mathcal{ZK}$ Protocol	Relation/Language	Ref.
$\mathcal{ZK}_{\textsc{Shuffle}}(\{\mathfrak{ct}_i\}, \{\mathfrak{ct}'_i\})$	$\mathcal{R}_{\textsc{Shuffle}} = \{(G, g, h, \{\mathfrak{ct}_i\}, \{\mathfrak{ct}'_i\}) \mid \exists \pi, \text{st.}$ ${C'_1}^{(i)} = g^{r_i} {C_1}^{(\pi(i))} \wedge {C'_2}^{(i)} = h^{r_i} {C_2}^{(\pi(i))} \wedge \pi \text{ is perm.}\}$	[11]
$\mathcal{ZK}_{\mathrm{EQ}}(\mathfrak{ct}_1, \mathfrak{ct}_2)$	$\mathcal{R}_{\mathrm{EQ}} = \{(G, g, h, \mathfrak{ct}_i = \langle \alpha_i, \beta_i \rangle_{i \in \{1,2\}}) \mid \exists (m_1, m_2), \text{st.}$ $\alpha_i = g^{r_i} \wedge \beta_i = m_i h^{r_i} \wedge m_1 = m_2\}$	[5]
$\mathcal{ZK}_{\textsc{no}}(\mathfrak{ct})$	$\mathcal{L}_{\textsc{no}} = \{(G, g, h, \mathfrak{ct} = \langle \alpha, \beta \rangle) \mid \exists (m_1 \neq 1), \text{st.}$ $\alpha = g^r \wedge \beta = m_1 h^r\}$	[13]

Table 1: List of zero-knowledge protocols used in our $\mathcal{ZK}_{\mathrm{EP}}$ protocol. Generator g and public key $h = g^{sk}$.

to k different outputs according to π, outputs the real ciphertexts followed by $k - 1$ dummy ciphertexts. This is repeated for each real ciphertext. The resulting output ciphertexts are all re-randomized. The dummy replacement step can be seen as a shuffling of the input ciphertexts. We use a proof of correct shuffle, $\mathcal{ZK}_{\textsc{Shuffle}}$, for correctness of this component.

- **Replication component:** This takes the output of the previous component as input. It directly outputs each real ciphertext but replaces each dummy ciphertext with an encryption of the real input that precedes it. At the end of this step, we have the necessary copies for each real input and the dummy inputs are eliminated. Naturally, all the ciphertexts are re-randomized. To prove correctness of this step, we need ZK proofs that the i-th output ciphertext has a plaintext equal to that of either the i-th input ciphertext or $(i - 1)$-th output ciphertext (these can be achieved using protocol $\mathcal{ZK}_{\mathrm{EQ}}$ defined in Table 1 as a building block). But this is not sufficient to guarantee a correct EP, as we also have to make sure that after the replication component there are no dummy ciphertexts left. For this, we assume that all dummy ciphertexts are encryptions of one. Then for each output ciphertext in the replication component we use a protocol $\mathcal{ZK}_{\textsc{no}}$, i.e. a ZK proof that the underlying plaintext is not one. The $\mathcal{ZK}_{\mathrm{Rep}}$ zero-knowledge protocol, is a compilation of three ZK protocols, two checking for equality of ciphertexts and one checking the inequality of plaintext to one.
- **Permutation component:** This takes the output of the replication component as input and permutes each element to its final location as prescribed by π. We again use the proof of correct shuffle, $\mathcal{ZK}_{\textsc{Shuffle}}$. for this component.

$\mathcal{ZK}_{\mathrm{EP}}$ *Protocol description* We assumed the inputs to the $\mathcal{ZK}_{\mathrm{EP}}$, to be the outputs of our encryption functionality. Prover applies the extended permutation to the ciphertexts $(\mathfrak{ct}_1, \ldots, \mathfrak{ct}_n)$, where $\mathfrak{ct}_i = (C_1^{(i)}, C_2^{(i)})$. The prover obtains a re-randomized $(\mathfrak{ct}'_1, \ldots, \mathfrak{ct}'_n)$, where $\mathfrak{ct}'_i = (C_1'^{(i)}, C_2'^{(i)})$. We employ the techniques of Cramer et al. [6], to combine HVZK proof systems corresponding to each component, at no extra cost, into HVZK proof systems of the same class for any (monotonic) disjunctive and/or conjunctive formula over statements proved in the component proof systems. Figure 7 shows the complete description of our

$\mathcal{ZK}_{\text{EP}}$ protocol. Note that we can choose dummy values from any set of random values S_d and substitute the $\mathcal{ZK}_{\text{NO}}(x)$ with $\vee_{\forall y \in S_d}(\mathcal{ZK}_{\text{EQ}}(x, y))$.

Protocol $\mathcal{ZK}_{\text{EP}}(\{\mathfrak{ct}_i\}, \{\mathfrak{ct}'_i\})$

Shared Input: Ciphertexts $(\mathfrak{ct}_1, \ldots, \mathfrak{ct}_n)$
P_1**'s Input:** Extended permutation π
P_1 **Evaluates the components.**

- Player P_1 finds the corresponding permutation π_1, and π_2 for Dummy-placement component and permutation components.
- P_1 applies the Dummy-placement component to $(\mathfrak{ct}_1, \ldots, \mathfrak{ct}_n)$, and rerandomizes to find $(\mathfrak{ct}_1^{(1)}, \ldots, \mathfrak{ct}_n^{(1)})$.
- P_1 applies the Replication component to $(\mathfrak{ct}_1^{(1)}, \ldots, \mathfrak{ct}_n^{(1)})$, and rerandomizes them to find $(\mathfrak{ct}_1^{(2)}, \ldots, \mathfrak{ct}_n^{(2)})$.
- P_1 applies the permutation component to $(\mathfrak{ct}_1^{(2)}, \ldots, \mathfrak{ct}_n^{(2)})$, and rerandomizes them to find $(\mathfrak{ct}'_1, \ldots, \mathfrak{ct}'_n)$.

P_1 **Computes the ZK proofs and sends everything**

- Player P_1 uses the $\mathcal{ZK}_{\text{Shuffle}}(\{\mathfrak{ct}_i\}, \{\mathfrak{ct}_i^{(1)}\})$ and $\mathcal{ZK}_{\text{Shuffle}}(\{\mathfrak{ct}_i^{(2)}\}, \{\mathfrak{ct}'_i\})$ protocols to produce proof of correctness for his evaluation of Dummy-placement component and permutation component.
- Player P_1 used the $\mathcal{ZK}_{\text{Rep}}(\{\mathfrak{ct}_i^{(1)}\}, \{\mathfrak{ct}_i^{(2)}\})$ to produce proof of correctness for his evaluation of Replication component as follows(using [6] for combination) (and $\mathcal{ZK}_{\text{Rep}}^1 = \mathcal{ZK}_{\text{NO}}\left(\mathfrak{ct}_1^{(2)}\right) \wedge \mathcal{ZK}_{\text{EQ}}(\mathfrak{ct}_1^{(1)}, \mathfrak{ct}_1^{(2)})$):
  - For $2 \le i \le n$:
$$\mathcal{ZK}_{\text{Rep}}^i = \left(\mathcal{ZK}_{\text{EQ}}(\mathfrak{ct}_i^{(1)}, \mathfrak{ct}_i^{(2)}) \vee \mathcal{ZK}_{\text{EQ}}(\mathfrak{ct}_{i-1}^{(2)}, \mathfrak{ct}_i^{(2)})\right) \wedge \mathcal{ZK}_{\text{NO}}\left(\mathfrak{ct}_i^{(2)}\right)$$
.
  - $\mathcal{ZK}_{\text{Rep}} = \wedge_{i=1,\ldots,n}(\mathcal{ZK}_{\text{Rep}}^i)$
- Player P_1 sends $(\mathfrak{ct}_1^{(1)}, \ldots, \mathfrak{ct}_n^{(1)})$, $(\mathfrak{ct}_1^{(2)}, \ldots, \mathfrak{ct}_n^{(2)})$, $(\mathfrak{ct}'_1, \ldots, \mathfrak{ct}'_n)$ and all proofs to other players.

Players verify P_1 **operations**

- Players verify P_1's operations by verifying the the proofs sent by P_1.

Fig. 7: The protocol for zero-knowledge proof of extended permutation.

Theorem 2. *The protocol described in Figure 7 is HVZK proof of an extended permutation π, $(\mathfrak{ct}_1, \ldots, \mathfrak{ct}_n)$ and $(\mathfrak{ct}'_1, \ldots, \mathfrak{ct}'_n)$ in the $\mathcal{ZK}_{\text{Shuffle}}$, $\mathcal{ZK}_{\text{EQ}}$, $\mathcal{ZK}_{\text{NO}}$ hybrid model, for the following relation:*

$$\begin{aligned}\mathcal{R}_{\text{EP}} = \{(G, g, h, \{\mathfrak{ct}_i\}, \{\mathfrak{ct}'_i\}) | \exists \pi, st.& \\ & {C'_1}^{(i)} = g^{r_i} {C_1}^{(\pi(i))} \wedge {C'_2}^{(i)} = h^{r_i} {C_2}^{(\pi(i))} \wedge \pi \textit{ is EP.}\}\end{aligned}$$

Proof. Refer to the full version [18] for proof.

Offline Protocol Having all the parts of the puzzle, we can give the complete $O(g)$ protocol for the offline phase. Figure 8 shows the description, with the proof of security given in full version [18].

Linear Implementation of Protocol $\mathcal{P}_{\text{OFFLINE}}$-Linear

The protocol is described in the $\mathcal{F}_{\text{MPC}}$-hybrid model, thus the only operation we need to specify is the **Input Function** one.

Input Function:

P_1 **Shares his Circuit/Function.**

- Player P_1 calls $(input, G_j)$ for all $j \in \{1, \ldots, g\}$.
- Players evaluate and open $[G_j] \cdot (1 - [G_j])$ for $j \in \{1, \ldots, g\}$. If any of them is not 0, players abort (since in this case P_1 has not entered a valid function).

Players Generate Randomness for inputs and outputs of EP.

- Players call $(random, \cdot)$ of $\mathcal{F}_{\text{MPC}}$ to generate shared random values for inputs $\boldsymbol{\ell} = ([\ell_1], \ldots, [\ell_{\mathsf{ow}(f)}])$ and outputs $([r_1], \ldots, [r_{\mathsf{iw}(f)}])$ of EP.
- Players call $(random, \cdot)$ of $\mathcal{F}_{\text{MPC}}$ to generate shared random values for the MAC value corresponding to inputs $\boldsymbol{t} = ([t_1], \ldots, [t_{\mathsf{ow}(f)}])$ and outputs $([s_1], \ldots, [s_{\mathsf{iw}(f)}])$ of EP.

P_1 **applies the EP to** $\boldsymbol{\ell}$ **and** $\boldsymbol{t}$**.**

- The players call KeyGen on the Enc_{Elg} functionality.
- The playes call $\mathsf{Encrypt}$ on the Enc_{Elg} functionality with the plaintexts $([\ell_1], \ldots, [\ell_{\mathsf{ow}(f)}])$ and the plaintexts $([t_1], \ldots, [t_{\mathsf{ow}(f)}])$, to obtain ciphertexts $\mathfrak{ct}_1, \ldots, \mathfrak{ct}_{\mathsf{ow}(f)}$ and $\mathfrak{ct}_1^{\dagger}, \ldots, \mathfrak{ct}_{\mathsf{ow}(f)}^{\dagger}$.
- Player P_1 applies the extended permutation to $(\mathfrak{ct}_1, \ldots, \mathfrak{ct}_{\mathsf{ow}(f)})$ and rerandomize to get $(\mathfrak{ct}'_1, \ldots, \mathfrak{ct}'_{\mathsf{ow}(f)})$, the same is done with $(\mathfrak{ct}_1^{\dagger}, \ldots, \mathfrak{ct}_{\mathsf{ow}(f)}^{\dagger})$ to obtain $(\mathfrak{ct}_1'^{\dagger}, \ldots, \mathfrak{ct}_{\mathsf{ow}(f)}'^{\dagger})$.
- Player P_1 uses the $\mathcal{ZK}_{\text{EP}}$ to prove that he has used a valid extended permutation.
- Players call the $\mathsf{Decrypt}$ on the Enc_{Elg} functionality (given in full version [18]) with ciphertexts $(\mathfrak{ct}'_1, \ldots, \mathfrak{ct}'_{\mathsf{ow}(f)})$ and $(\mathfrak{ct}_1'^{\dagger}, \ldots, \mathfrak{ct}_{\mathsf{ow}(f)}'^{\dagger})$ so as to obtain $([\ell_{\pi(1)}], \ldots, [\ell_{\pi(\mathsf{ow}(f))}])$ and $([t_{\pi(1)}], \ldots, [t_{\pi(\mathsf{ow}(f))}])$.

Players Compute p_i, q_i**.**

- For $i \in \{1, \ldots, \mathsf{iw}(f)\}$ players call $\mathcal{F}_{\text{MPC}}$ to compute:
  $[p_i] = [r_i] - [\ell_{\pi(i)}] \doteq [r_i - \ell_{\pi(i)}]$, $[q_i] = [s_i] - [t_{\pi(i)}] + p_i \cdot [K] \doteq [s_i - t_{\pi(i)} + p_i \cdot K]$

Fig. 8: The protocol for linear implementation of the Offline Phase

5 A practical Implementation of $\mathcal{F}_{\text{Offline}}$ with $O(g \cdot \log g)$ Complexity

A $O(g \cdot \log g)$ protocol to implement $\mathcal{F}_{\text{OFFLINE}}$ is given in full version [18], and is in the $\mathcal{F}_{\text{MPC}}$-hybrid model. Following the ideas in [17], we implement the functionality via secure evaluation of a *switching network* corresponding to the mapping π_f.

Switching Networks. A switching network SN is a set of interconnected switches that takes N inputs and a set of selection bits, and outputs N values. Each *switch* in the network accepts two ℓ-bit strings as input and outputs two ℓ-bit strings. In this paper we need to use a switching network that contains two switch types. In the first type (*type 1*), if the selection bit is 0 the two inputs remain intact and are directly fed to the two outputs, but if the selection bit is 1, the two input values swap places. In the second type (*type 2*), if the selection bit is 0, as before, the inputs are directly fed to outputs but if it is 1, the value of the first input is used for both outputs. For ease of exposition, in our protocol description we assume that all switches are of type 1, but the protocol can be easily extended to work with both switch types.

The *mapping* $\pi : \{1 \dots N\} \rightarrow \{1 \dots N\}$ corresponding to a switching network SN is defined such that $\pi(j) = i$ if and only if after evaluation of SN on the N inputs, the value of the input wire i is assigned to the output wire j (assuming a standard numbering of the input/output wires). In [17] it is shown how to represent any mapping with a maximum of N inputs and outputs via a network with $O(N \cdot \log N)$ type 1 and 2 switches (We refer the reader to [17] for the details). This yields a switching network with $O(g \cdot \log g)$ switches to represent the mapping for a circuit with g gates.

High Level Description. It is possible to implement the $\mathcal{F}_{\text{OFFLINE}}$ by securely computing a circuit for the above switching network using the $\mathcal{F}_{\text{MPC}}$. But for all existing MPC that meet our requirements, this would require $O(\log g)$ rounds of interaction which is the depth of the circuit corresponding to the switching network. We show an alternative constant-round approach with similar computation and communication efficiency. It follows the same idea as the OT-based protocol of [17] where the OT is replaced with an equivalent functionality implemented using $\mathcal{F}_{\text{MPC}}$. The main challenge in our case is to achieve *active security* and in particular to ensure that P_1 cannot cheat in his local computation. We do so by checking P_1's actions using one-time MACs of the values he computes on, and allow the other parties to learn his input and proceed without him, if he is caught cheating (or aborting).

Next we give an overview of the protocol. The protocol has four main components (as described in full version [18]). In the first step, P_1 converts his mapping π to selection bits for the switching network (i.e. b_is) and shares them with all players. He also shares a bit G_i indicating the function of gate i, with other players. In the second step, players generate random values for every wire in the network. P_1, based on his selection bit for the switch, learns two of the four possible "subtractions" of the random values for two output wires from those of the input wires i.e. $u_0^{\ell,i}$ and $u_1^{\ell,i}$. A similar process is performed for the t values to obtain $u_0^{t,i}$ and $u_1^{t,i}$ (Figure 9 shows this process in a diagram). These subtractions enable P_1 to transform a pair of values blinded with the random values of input wires, to the same pair of values permuted (based on the selection bit) and blinded with the random values of the output wires. All of the above can be implemented using the operations provided by the $\mathcal{F}_{\text{MPC}}$.

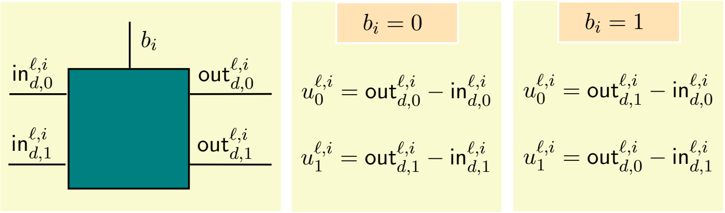
b_i
$\mathsf{in}^{\ell,i}_{d,0}$
$\mathsf{in}^{\ell,i}_{d,1}$
$\mathsf{out}^{\ell,i}_{d,0}$
$\mathsf{out}^{\ell,i}_{d,1}$
$b_i = 0$
$u^{\ell,i}_0 = \mathsf{out}^{\ell,i}_{d,0} - \mathsf{in}^{\ell,i}_{d,0}$
$u^{\ell,i}_1 = \mathsf{out}^{\ell,i}_{d,1} - \mathsf{in}^{\ell,i}_{d,1}$
$b_i = 1$
$u^{\ell,i}_0 = \mathsf{out}^{\ell,i}_{d,1} - \mathsf{in}^{\ell,i}_{d,0}$
$u^{\ell,i}_1 = \mathsf{out}^{\ell,i}_{d,0} - \mathsf{in}^{\ell,i}_{d,1}$

Fig. 9: The i-th switch. (superscripts: label of value subject to permute (ℓ or t), and switch index i) (subscripts: d refers to data, m refers to MAC, wire index 0 denotes the top wire in switch and 1 the bottom wire in switch)

In the third step, P_1 obtains the blinded ℓ and t values where the blinding for each is the random value for the corresponding input wire to the network (these are $h^{\ell,i}_d, h^{t,i}_d$, etc). Party P_1 can now process each switch as discussed above using the subtraction values in order to evaluate the entire network. At the end of this process, P_1 holds blinded values of the outputs of the switching network (blinded with randomness of the output wires).

In the final step, parties check that P_1 has not cheated during his evaluation, since he performed this step locally and not through the $\mathcal{F}_{\text{MPC}}$ operations. We use one-time MACs to achieve this goal. In particular, besides mapping blinded values through the network, P_1 also maps the corresponding one-time MACs (generated using the fixed-key K). This is done using a similar process described above and via the $v^{\ell,i}_j, v^{t,i}_j$ values. At the end of this process, P_1 holds one-time MACs for the blinded outputs of the switching network, in addition to the values themselves. Players then use the MPC functionality to jointly verify that the MACs indeed verify the values P_1 shared with them (i.e. $n^{\ell,i}$ and $m^{\ell,i}$ are the same, etc). As a result, P_1 can only cheat by forging the MACs which only happens with a negligible probability. If the MACs pass, parties compute and open the "difference vectors" by subtracting the mapped ℓ and t-value vectors from the r and s-value vectors. Refer to full version [18] for more details. If one instantiates the $\mathcal{F}_{\text{MPC}}$ by SPDZ [8], which has the $m.\log(p^k)$ complexity, then our complexity would be $m\,(10(2g\log 2g - 2g + 1) + 4g)\,.\log(p^k)$. Refer to full version [18] for the proof of the following theorem.

Theorem 3. *In the $\mathcal{F}_{\text{MPC}}$-hybrid model the protocol $\mathcal{P}_{\text{OFFLINE}}$ in full version [18] securely implements the functionality in Figure 2, with complexity $O(g \cdot \log g)$.*

6 Acknowledgements

This work has been supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant EP/I03126X, and by Defense Advanced

Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079.

The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

1. M. Abadi and J. Feigenbaum. Secure circuit evaluation. *J. Cryptology*, 2(1):1–12, 1990.
2. M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider. Secure evaluation of private linear branching programs with medical applications. In M. Backes and P. Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 424–439. Springer, 2009.
3. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.
4. J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel. Privacy-preserving remote diagnostics. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 498–507. ACM, 2007.
5. D. Chaum and T. P. Pedersen. Wallet databases with observers. In *CRYPTO*, pages 89–105, 1992.
6. R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Y. Desmedt, editor, *Advances in Cryptology - CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer Berlin Heidelberg, 1994.
7. I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography: PKC '09*, Irvine, pages 160–179, Berlin, Heidelberg, 2009. Springer-Verlag.
8. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [25], pages 643–662.
9. I. Damgård and S. Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *Theory of Cryptography*, pages 621–641. Springer, 2013.
10. J. Furukawa. Efficient and verifiable shuffling and shuffle-decryption. *IEICE Transactions*, 88-A(1):172–188, 2005.
11. J. Furukawa and K. Sako. An efficient scheme for proving a shuffle. In J. Kilian, editor, *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 368–387. Springer Berlin Heidelberg, 2001.
12. R. Gennaro, C. Hazay, and J. S. Sorensen. Text search protocols with simulation based security. In P. Q. Nguyen and D. Pointcheval, editors, *Public Key Cryptography*, volume 6056 of *Lecture Notes in Computer Science*, pages 332–350. Springer, 2010.

13. C. Hazay and K. Nissim. Efficient set operations in the presence of malicious adversaries. In *Public Key Cryptography*, pages 312–331, 2010.
14. Y. Ishai and A. Paskin. Evaluating branching programs on encrypted data. In S. P. Vadhan, editor, *TCC*, volume 4392 of *Lecture Notes in Computer Science*, pages 575–594. Springer, 2007.
15. J. Katz and L. Malka. Constant-round private function evaluation with linear complexity. In D. H. Lee and X. Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 556–571. Springer, 2011.
16. V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In G. Tsudik, editor, *Financial Cryptography*, volume 5143 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2008.
17. P. Mohassel and S. Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 557–574. Springer, 2013.
18. P. Mohassel, S. Sadeghian, and N. P. Smart. Actively secure private function evaluation. Cryptology ePrint Archive, Report 2014/102, 2014. `http://eprint.iacr.org/`.
19. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [25], pages 681–700.
20. S. Niksefat, B. Sadeghiyan, P. Mohassel, and S. Sadeghian. Zids: A privacy-preserving intrusion detection system using secure two-party computation protocols. *The Computer Journal*, 2013.
21. R. Ostrovsky, A. Paskin-Cherniavsky, and B. Paskin-Cherniavsky. Maliciously circuit-private fhe. Cryptology ePrint Archive, Report 2013/307, 2013. `http://eprint.iacr.org/`.
22. A. Paus, A.-R. Sadeghi, and T. Schneider. Practical secure evaluation of semi-private functions. In M. Abdalla, D. Pointcheval, P.-A. Fouque, and D. Vergnaud, editors, *ACNS*, volume 5536 of *Lecture Notes in Computer Science*, pages 89–106, 2009.
23. R. Raz. Elusive functions and lower bounds for arithmetic circuits. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pages 711–720, New York, NY, USA, 2008. ACM.
24. A.-R. Sadeghi and T. Schneider. Generalized universal circuits for secure evaluation of private functions with application to data classification. In P. J. Lee and J. H. Cheon, editors, *ICISC*, volume 5461 of *Lecture Notes in Computer Science*, pages 336–353. Springer, 2008.
25. R. Safavi-Naini and R. Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.
26. L. Valiant. Universal circuits (preliminary report). In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 196–203. ACM, 1976.