Equivocating Yao: Constant-Rounds Adaptively Secure Multiparty Computation in the Plain Model

Ran Canetti[*] Oxana Poburinnaya[†] Muthuramakrishnan Venkitasubramaniam[‡]

December 30, 2016

Abstract

Yao's garbling scheme is one of the basic building blocks of crytographic protocol design. Originally designed to enable two-message, two-party secure computation, the scheme has been extended in many ways and has innumerable applications. Still, a basic question has remained open throughout the years: Can the scheme be extended to guarantee security in the face of an adversary that corrupts both parties, adaptively, as the computation proceeds?

We answer this question in the affirmative. We define a new type of encryption, called functionally equivocal encryption (FEE), and show that when Yao's scheme is implemented with an FEE as the underlying encryption mechanism, it becomes secure against such adaptive adversaries. We then show how to implement FEE from any one way function.

Combining our scheme with non-committing encryption, we obtain the first two-message, two-party computation protocol, and the first constant-round multiparty computation protocol, in the plain model, that are secure against semi-honest adversaries who can adaptively corrupt all parties. A number of extensions and applications are described within.

Keywords: Adaptive Security, Garbled Circuits, Secure Computation

[*]Boston University and Tel Aviv University. Email: `canetti@bu.edu`

[†]Boston University. Email: `oxanapob@bu.edu. Research was done while visiting Cornell Tech.`

[‡]University of Rochester. Email `muthuv@cs.rochester.edu.`

Contents

1 Introduction

Secure multi-party computation (MPC) protocols allow a set of mutually distrustful parties to engage in a joint computation for evaluating an agreed-upon function of their local inputs, while preserving the privacy of their inputs up to what is revealed by the function. First envisioned by Yao [Yao82], and realized in a sequence of breakthrough protocols starting with [Yao86, GMW87, BGW88, CCD88, GL90], the concept evolved into a vibrant discipline that is one of the standard bearers of cryptography. Indeed, today we have a mature theory of secure distributed computation, with highly optimized and innovative protocols in a variety of models and settings. We also have a number of real-world systems, both academic and commercial, that use this rich theory to better society.

Still, even today, we are unable to provide satisfactory answers to some very basic challenges in secure computation. One of these challenges is providing adequate security and efficiency guarantees in the natural setting where protocol participants become corrupted during the course of the computation, and furthermore the identities of the corrupted parties are determined adversarially and adaptively as the computation unfolds. In particular, while security in this setting (traditionally called adaptive security) has been extensively studied, the following very basic question has remained unanswered:

How many rounds are required for adaptively secure multiparty computation?

We first focus on the very basic case of two parties with honest-but-curious corruptions. Even here there is a large gap between the best-known protocols and what is potentially possible: In the static case (where parties are either compromised from the start or remain compromised throughout), the Yao garbled circuit protocol [Yao86, Rog91] provides a classic two-message solution with no trusted setup other than authenticated communication. The protocol can be easily extended to withstand adaptive corruptions in settings where parties can effectively erase sensitive local information. It can also be easily extended to withstand adaptive corruption of *one* of the parties even without having to trust effective erasure of data, e.g. by encrypting the communication using non-committing encryption [CFGN96]. However, in the pertinent case where local data cannot be erased in a trustworthy way, and where both parties can be eventually compromised, the best known solution so far that does not use additional trusted setup takes $O(d)$ rounds, where d is the depth of the evaluated circuit [GMW87, CLOS02a]. The case where both parties are eventually compromised is of interest in settings where the analyzed protocol is a component in a larger system and one wants to provide security guarantees for the system even when both participants are compromised. It is also instrumental in providing leakage resilience guarantees, as discussed in more detail later on.

The situation is similar in the multi-party setting: With static corruptions, as well as adaptive corruptions with erasures, and all-but-one adaptive corruptions without erasures, we have constant rounds solutions [BMR90, IPS08, GS12]. However, when all parties can eventually be corrupted and trustworthy erasure is not available, the $O(d)$-rounds protocol of [GMW87, CLOS02a] is essentially the best known.

The same dichotomy translates also to the case of Byzantine corruptions in the plain model with respect to non-concurrent security [Can00, Gol04].

We note that in the common reference string model (namely, when the parties are given access to a common string that was sampled by a trusted party from a predetermined special distribution) we do have constant rounds adaptively secure two-party and multi-party computation protocols that do not need trustworthy data erasure and withstand eventual corruption of all parties and Byzantine fault [GP14, DKR14, CGP15, CP16]. In fact these protocols are even UC-secure. On the down side, all of these protocols are based the heavy machinery of indistinguishability obfuscation.

1.1 Our results

We show constant-rounds, adaptively secure protocols in all the above cases. Our protocols use minimal hardness assumptions - analogous assumptions to those needed for obtaining static security. Specifically:

Theorem (informal): *Assume existence of non-committing encryption schemes. Then there exist:*

- *A* minimum interaction *(i.e., two-message) two-party general function evaluation protocol that withstands adaptive honest-but-curious corruption of both parties. The protocol is in the plain model and does not use data erasures.*
- *A* constant-round *multiparty general function evaluation protocol in the plain model that withstands adaptive honest-but-curious corruption of all parties.*

Assuming in addition collision resistant hashing and dense cryptosystems, there exist:

- *A constant-round multiparty general function evaluation protocol in the plain model that withstands adaptive* Byzantine *corruption of all parties, in the non-concurrent security model [Can00].*
- *A constant-round* UC-secure *multiparty general function evaluation protocol in the common random string model, in face of adaptive Byzantine corruption of all parties.*

Application to leakage tolerant computation. [GJS11, BCH12]. A more nuanced (and considerably stronger) level of security for multi-party computation protocols considers adversaries who, in addition to corrupting parties, can obtain (presumably via side channel attacks) some partial information on the internal states of *all* parties. Still, this information is obtained from each party individually. Security against such attacks requires that, for any l, any adversary that learns vector of values of some function with l-bit output, applied, individually, to the state of each party, can be simulated "in the ideal model" given only l bits of information on the input and output of *each party individually*. Thus, the notion of leakage tolerance provides "graceful degradation" guarantees, bounding the rate of degradation of security with the increase in leakage[1].

It is currently known how to realize only weaker variants of this strong requirement. One significantly weaker variant only requires that the leakage from the state of each individual party can be simulated given the inputs and outputs of *all parties*, pooled together [BGJ+13]. Another variant allows corruption of only some of the parties [BDL14].

We show that a variant of our multi-party protocol provides the first leakage-tolerant protocol that tolerates any level of leakage from any number of parties, and where the information learned in each leakage operation is simulatable given only the inputs and outputs of the victim party alone. As in [BDL14], our protocol requires an input-independent leak-free preprocessing stage. However, while in [BDL14] the preprocessing stage suffices for evaluating the function on multiple inputs, our preprocessing stage suffices only for a single evaluation. In contrast, while [BDL14] work in the common reference string model, our protocol works in the plain model. This also gives the first compiler of general circuits to two-component leakage resilient circuits in the Only Computation Leaks (OCL) model [MR04, GR10] — albeit with the caveat that the offline preprocessing stage suffices only for a bounded number of evaluations. The currently best solution requires a polynomial number of components [GR10]:

[1]Leakage tolerance should not be confused with a related but different notion of a leakage resilience, which states that the adversary shouldn't learn anything about the secret - say, about the message in case of encryption - even if several bits of the secret key are leaked.

Theorem (informal): *If there exist non-committing encryption schemes then there exists a multi-party leakage tolerant general function evaluation protocol in the plain model. The protocol consists of an off-line leak-free stage, followed by an on-line stage where leakage can be obtained from all parties. Both stages take a constant number of rounds.*

1.2 Our Techniques

Yao's two-party protocol. We first recall Yao's protocol for two-party circuit evaluation. Yao's protocol consists of two main components: a *garbling scheme* and *oblivious transfer (OT)*. Recall that a garbling scheme allows to transform a circuit C and an input x into their garbled versions $\tilde{C}, \tilde{x}$. Given $\tilde{C}$ and $\tilde{x}$, it should be possible to compute $y = C(x)$. On the other hand, security guarantees that $\tilde{C}, \tilde{x}$ do not reveal anything about x (except for what is revealed by the output y). This is formalized by requiring that the simulator produces good-looking garbled values $\tilde{C}$ and $\tilde{x}$, given only C and y (but not x). To be useful in Yao two-party protocol, the garbling scheme needs an additional property called *bit-decomposability*, which states that it should be possible to garble each input bit separately, i.e. without knowing other input bits nor the circuit.

The Yao protocol for evaluating $C(x_1, x_2)$ for a public circuit C then proceeds as follows. One of the parties (the garbler G) generates the garbled circuit $\tilde{C}$ and its own garbled input $\tilde{x}_1$ and sends $\tilde{C}, \tilde{x}_1$ to the other party (the evaluator E). To enable evaluation $\tilde{C}(\tilde{x}_1, \tilde{x}_2)$, E should also get $\tilde{x}_2$; however, G doesn't know x_2 and cannot garble it directly. Instead, G sends $\tilde{x}_2$ to E via OT: For each position $i = 1, \dots, |x_2|$, G garbles both input value 0 and and input value 1. Next, G lets E learn exactly one of the two garbled bits for each input location i. E chooses to take the bit which corresponds to its input value for that location. After E receives $\tilde{x}_2$, it can evaluate the garbled circuit $\tilde{C}(\tilde{x}_1, \tilde{x}_2)$ and learn the output y. Since OT can be implemented in two messages (one message from E to G, and then one message from G to E), the resulting protocol requires only two messages.

Static security of this protocol (i.e. security against either an a-priori corrutped G or an a-priori corrupted E) follows from security of the garbling scheme and the OT.

Specifically, if the garbler is corrupted, the simulator learns the garbler's input x_1 and generates the garbled circuit and garbled input honestly. If the evaluator is corrupted, or if nobody is corrupted, the simulator shows the simulated garbled circuit and simulated garbled input.

The challenge of adaptive security. Recall that in the setting of adaptive security, the adversary can corrupt parties as the protocol proceeds; upon each corruption, the adversary learns the whole internal state (e.g. inputs and random coins) of that party. In the ideal world the simulator obtains only the input and output of the corrupted party and has to produce consistent random coins of that party. Furthermore, inputs and outputs are learnt only at the time of corruption.

The above static-corruptions simulation of the Yao protocol fails in in the adaptive setting, even if ideally secure communication is provided. To illustrate the problem, consider the adversary that waits until the protocol is finished, then corrupts the evaluator E, and then the garbler G. Upon corruption of E, the simulator $\mathcal{S}$ is given E's input x_2 and output y and is required to present the garbled circuit and both garbled inputs; however, $\mathcal{S}$ doesn't know G's input x_1 at this point, and therefore can only present the *simulated* garbled circuit and inputs. Upon corruption of G, however, the adversary expects to see G's internal state - and in particular randomness which was used to garble the circuit and inputs. Now $\mathcal{S}$ is in trouble: not only does it have to convince that the (simulated) garbled circuit was generated honestly - which is already hard for $\mathcal{S}$ to do - it also needs to make sure that the simulated garbled input $\tilde{x}_1$ looks like a garbling of the value x_1, which $\mathcal{S}$ just learned.

Equivocal garbling schemes. We get around this difficulty by constructing a scheme that allows the simulator to generate "fake garbled circuits" and "fake garbled inputs" that can be later consistently "opened" (by presenting consistent randomness of the garbling) to a given input x. We call such garbling schemes *equivocal*. In a bit more detail, an equivocal garbling scheme allows the simulator to first generate a garbled circuit $\hat{C}$ together with garbled input $\hat{x}$, given only C and $y = C(x)$. Later, given x, the simulator generates a fake randomness of the garbling that makes $\hat{C}, \hat{x}$ look like a real garbling of C, x. [2]

Our equivocal garbling scheme can then be used in a straightforward way, together with adaptively secure OT and non-committing encryption, to obtain our first main result, namely adaptively secure, two-message, two-party computation.

Our equivocal garbling scheme is a modification of the traditional Yao garbling scheme; thus, we first recall how the latter works. Given a public circuit C and an input x, for each wire w in C, the garbler chooses two random labels, k_w^0 and k_w^1, where each label is a λ-bit string, λ being the security parameter. Then for each gate g in C, the garbler prepares four ciphertexts $c_g^{00}, c_g^{01}, c_g^{10}, c_g^{11}$, where $c_g^{b_1,b_2}$ is an encryption of $k_{w_3}^{b_3}$ under a combination of the keys $k_{w_1}^{b_1}$ and $k_{w_2}^{b_2}$, where w_1, w_2 are the input wires to g, w_3 is the output wire of g, and $b_3 = g(b_1, b_2)$ is the value of the output bit of gate g on input bits b_1, b_2 (there are several standard ways to implement the underlying encryption mechanism using any one way function). Output gates encrypt output bits instead of labels.

Each garbled gate consists of the four ciphertexts listed in random order, and the garbled circuit $\tilde{C}$ consists of all garbled gates. The garbled input $\tilde{x}$ consists of labels for input wires corresponding to bits of x, i.e. $k_i^{x_i}$ for every $i = 1, \ldots, |x|$. Given garbled circuit and garbled input, the circuit can be evaluated gate by gate, by decrypting an appropriate ciphertext and learning the label for its output wire.

This scheme satisfies the standard (i.e. non-equivocal) definition of the garbling scheme. Indeed, to simulate the garbled circuit given C and y, the simulator, instead of encrypting k_w^0, k_w^1 for each gate, will encrypt the same random label k_w four times (output gates should instead encrypt y). The input is garbled by giving k_i for $i = 1, \ldots, |x|$. Intuitively, this simulation is good, since evaluation results in y, and since the adversary can only decrypt one ciphertext per gate, which decrypts to a random label, just like in the real case. However, this scheme is not equivocal: if the simulator has to explain how ciphertexts were generated, then it has to show randomness of encryption and all keys, but in this case the adversary would see that all four ciphertexts encrypt the same label.

This problem would be solved if the simulator was able to pretend that a ciphertext c, encrypting k_w, actually encrypts a different value $\bar{k}_w$. And indeed, the first attempt to solve this problem may be to use non-committing encryption (NCE) for generating the four ciphertexts that comprise each garbled gate. (Recall that NCE allows the simulator to generate "dummy ciphertexts" c that can be later opened to any message m in some domain. In the case of symmetric encryption, which suffices here, this means demonstrating a dummy ciphertext c and then, given a message m, demonstrating a key k_m and random input r_m for the encryptor such that $Enc(m, k_m, r_m) = c$ and $Dec(k_m, c) = m$.)

If the garbling scheme is instantiated with NCE, it indeed becomes equivocal: roughly, the simulator can generate dummy ciphertexts first, but "open" them appropriately later, so that they appear encrypting k_w, k_w, k_w, and $\bar{k}_w$. However, we know that for NCE the key size must be at least the message size [Nie02]. Since each label for an input wire to a gate is used to encrypt *two* out of four ciphertexts, we have that with NCE the labels for the input wire to a gate must be at least *twice as long* as the labels for the output wire. This means that circuits of at most logarithmic depth can be garbled in polynomial time. (In fact, for such circuits the one-time-pad provides a perfect NCE with statistical security.)

A number of attempts to save on the length of NCE keys, at the price of limiting the equivocation capabilities, have been proposed, although in a different context [HJO+16], [GWZ09]. We note that neither of these methods seem to suffice in our setting. We give more details at the end of the introduction in section 1.3.

[2] We do not use the term *adaptive garbling* since this term has already been used in the literature to denote a very different form of adaptivity for garbling schemes [GKR08, BHR12]. See section 1.3 for details.

Functionally equivocal encryption (FEE). We avoid this exponential blowup in label size by using a new type of symmetric encryption scheme, which we call functionally equivocal encryption (FEE). FEE behaves much like symmetric NCE, except that the keys are significantly shorter — at the price of somewhat restricted equivocation capabilities. That is, consider the case where the plaintext space is large ($\{0,1\}^l$), but we want to "open" dummy ciphertexts to messages only from a much smaller, but still exponential, subset $R \subset \{0,1\}^l$, where $|R| = 2^n$, and $n << l$. (Say, R may be the set of English sentences, or the range of a pseudorandom generator from λ bits to 2λ bits.) If NCE is used, keys still have to be as long as l; in FEE length of keys instead depends on n which is much shorter than l. It turns out that such a scheme can be constructed for any set R which has a short description - that is, for which there exists an efficient circuit $f : \{0,1\}^n \to R$ which enumerates its elements.

At a high level, the syntax of FEE is the following: an encryption algorithm can encrypt any message $m \in \{0,1\}^l$ (for technical reasons, encryption also needs to know parameters of f, i.e. its description size, input size, and output size). The simulator can fix a set R, which it wants to equivocate to, by choosing its description function F, and simulate a dummy ciphertext $c \leftarrow \mathsf{Sim}(f)$. Later the simulator can open c to any $m' \in R$, as long as it knows preimage x such that $f(x) = m'$, by running $(r_{\mathsf{Enc}}, k) \leftarrow \mathsf{Sim}(c, x, f)$. However, for our garbling scheme we need a slightly different syntax, as we describe below.

More formally, an instance of an FEE scheme is parameterized by the length parameters s, n, l, where $\{0,1\}^l$ is the message space, n is equivocality parameter (i.e. the size of set R we want to equivocate to is 2^n), and s is the size of description of the function $f : \{0,1\}^n \to R$, which defines the set. In addition to standard key generation, encryption and decryption algorithms, there is a simulator that operates in three steps, where each step involves a different algirithm as follows.

First, the simulator uses algorithm SimEnc to generate dummy ciphertexts. Let f be a function from n bits to l bits with description size s. Algorithm SimEnc takes as input a description of this function and generates a dummy ciphertext c_f together with a trapdoor.

Then, the simulator uses algorithm Equiv to generate a dummy (symmetric) key. Algorithm Equiv takes the trapdoor and a value $x \in \{0,1\}^n$ and generates a dummy key k_x such that $Dec(k_x, c_f) = m$, where $m = f(x)$.

Finally, the simulator uses algorithm Adapt to generate dummy randomness for the encryption process. Algorithm Adapt takes the trapdoor, x, and k_x, and outputs dummy randomness r_x such that $Enc(k_x, m, r_x) = c_f$, where $m = f(x)$.

The values k_x, r_x, c_f should be distributed indistinguishably from a real key, real randomness and real ciphertext in the process of encryption and decryption of m. In particular, the function f and the value x should remain hidden even given m, k_x, r_x, c_f. We also stress that the function f is used only in the generation of dummy values. Real encryption and decryption works for arbitrary messages in $\{0,1\}^l$, and does not need to know f.

Importantly, besides security we also require efficiency; that is, FEE keys should have size $n \cdot \mathsf{poly}(\lambda)$ - potentially much shorter than NCE keys for l-bit messages, which size has to be at least l. This efficiency requirement will make sure that the size of labels (i.e. FEE keys) in the garbled circuit doesn't grow with the depth of the circuit, thus allowing to garble any polynomial-sized circuit C. At the same time, we will be able to prove security of the garbling, given this limited equivocation, as we describe below.

From FEE to equivocal garbling. Our equivocal garbling scheme uses FEE as the underlying encryption mechanism. The garbling process is the standard one: the garbler chooses FEE keys and generates garbled gates as double-encryptions of the next level keys (the difference is that FEE encryption needs to know parameters of the function f to be used in equivocation; these parameters are some fixed polynomials in $|C|, \lambda$, and $|x|$). The simulation however is done differently: the idea is to have the adaptive simulator $\mathcal{S}$ for the garbling scheme choose the functions for the different dummy ciphertexts so that it can later equivocate the keys, plaintexts and randomness to complete the simulation.

This is done as follows. At the first stage of the simulation (i.e. when the simulator has to produce $\tilde{C}, \tilde{x}$, given C and y), the simulator chooses at random one label k_w for each wire in the circuit C; these labels will be the *active* labels, namely the wire labels that are exposed to the adversary at this stage. Next, the simulator computes the simulated garbled gates, in sequence, gate by gate from the output wire of the circuit to the input wires, in topological order (we remark that this sequentiality is imposed for exposition purposes only not crucial, we later show how to garble all gates in parallel.)

Let g be a gate with input wires w_1, w_2 and output wire w_3. The four ciphertexts $c_g^{00}, ..., c_g^{11}$ are computed, using an FEE scheme, as follows. The simulator chooses a random index among $00, 01, 10, 11$, say, 00. Then c_g^{00} is set to be a real FEE encryption of k_{w_3} with keys k_{w_1} and k_{w_2} (the output gate instead encrypts y). The other three ciphertexts $c_g^{01}, c_g^{10}, c_g^{11}$ are dummy ciphertexts, created using the SimEnc procedure of the FEE scheme with respect to special functions f^{01}, f^{10}, f^{11}, which will be explained later and which will help the simulator in equivocation.

The simulator presents these FEE ciphertexts as the garbled circuit, and gives $k_1, \ldots, k_{|x|}$ as the garbled input.

At the second stage of the simulation, i.e. when the simulator has to present randomness used to garble, the simulator first will set each inactive key to be $\bar{k}_w = \mathsf{Equiv}(x, td_w)$, and give it to the adversary. Thus the adversary now possesses all keys, both active and inactive, and therefore can decrypt all ciphertexts and check whether the gates were garbled correctly. We construct functions f such that the gates will indeed appear correct to the adversary: namely, each gate will encrypt one key 3 times and the other key once (assuming a NAND gate), and furthermore, one of these keys will exactly be k_{w_3}, and the other key will be $\bar{k}_{w_3}$ (where w_3 is an output wire of the gate). Finally, it will be consistent with the computation, in particular, if w_3 gets assigned 1 in the computation $C(x)$, then k_{w_3} appears in 3 ciphertexts of a NAND gate; it w_3 is 0, then only once. To achieve this, we define functions $f^{b_1 b_2}$ as follows (with hardwired values b_1, b_2, C, wire indices w_1, w_2, w_3, the active labels $k_{w_1} k_{w_2} k_{w_3}$, and a trapdoor value td_{w_3} that comes from the FEEs associated with the gate g' that takes wire w_3 as input.):

Given input x, do:

- Evaluate $C(x)$ and find the bit assignments $\sigma_1, \sigma_2, \sigma_3$ to wires w_1, w_2, w_3, respectively.
- Associate the label k_{w_3} (hardwired) with bit σ_3. Compute $\bar{k}_{w_3} = \mathsf{Equiv}(x, td_{w_3})$ and associate it with the bit $1 - \sigma_3$ (We call $\bar{k}_{w_3}$ the *inactive label* for wire w_3.)
- Return a label according to the logical value of the gate. That is, if the gate is a NAND gate and $(b_1 \oplus \sigma_1) \,\mathsf{NAND}\, (b_2 \oplus \sigma_2) = \sigma_3$ then output the active label k_{w_3}. Else, output the inactive label $\bar{k}_{w_3}$.

To illustrate why the gate looks like a normal Yao garbled gate under keys $k_{w_1}, \bar{k}_{w_1}, k_{w_2}, \bar{k}_{w_2}$, consider the example where w_1 was assigned 0, w_2 was assigned 1, and w_3 was assigned $0 \,\mathsf{NAND}\, 1 = 1$ by the computation $C(x)$. Thus, the job of the simulator is to open four ciphertexts so that they encrypt k_{w_3} three times and $\bar{k}_{w_3}$ only once, since active key k_{w_3} should look like a label for 1. To simplify things, let us ignore the fact that ciphertexts are double encryptions; let us pretend that all four ciphertexts are only encrypted once - under k_{w_1} and $\bar{k}_{w_1}$. Since our goal is to demonstrate how inactive key $\bar{k}_{w_1}$ will decrypt things correctly, we focus on ciphertexts which are encrypted under this key, namely, c^{10} and c^{11}.(Recall that c^{00} will be decrypted correctly to k_{w_3} under k_{w_1}, k_{w_2}, since it was an honest encryption. The other ciphertext, c^{01}, is a dummy ciphertext under keys k_{w_1} and $\bar{k}_{w_2}$. The inactive key $\bar{k}_{w_2}$ will make sure it decrypts appropriately, using a mechanism similar to described above.)

First let's see how these two ciphertexts *should* be decrypted: since c^{00} is the active ciphertext corresponding to wire assignments $0, 1$, c^{10} should correspond to $1, 1$, (indeed, c^{00} and c^{10} are ciphertexts for the opposite bits of w_1, but the same bit of w_2) and thus it should pretend to encrypt the key for

$1\,\mathsf{NAND}\,1 = 0$, i.e. $\bar{k}_{w_3}$. Similarly, c^{11} should correspond to $1, 0$, and thus it should pretend to encrypt the key for $1\,\mathsf{NAND}\,0 = 1$, i.e. active key k_{w_3}.

Now let's compare to how these ciphertexts *will* be decrypted. Since c^{10} was generated under the function f^{10}, decrypting it with $\bar{k}_{w_1} = \mathsf{Equiv}(x)$ will result in $f^{10}(x)$; c^{11} will decrypt to $f^{11}(x)$. A closer look at these functions reveals that $f^{10}(x)$ and $f^{11}(x)$ are exactly $\bar{k}_{w_3}$ and k_{w_3}, as it should be. Indeed, each f takes hardcoded active k_{w_3}, computes inactive key $\bar{k}_{w_3}$, and decides which to output, using, in fact, exactly the same reasoning as the one we used above to decide which key should be the output!

While in the actual proof we have to deal with slightly more complicated functions due to the fact that each $c^{b_1 b_2}$ is a double encryption, the idea is exactly the same: let functions f evaluate $C(x)$ and themselves decide, what to output.

It remains for the simulator to explain randomness of encryption; this is done by running the Adapt algorithm of the FEE on input x, along with the appropriate keys and trapdoors.

Finally, we comment on the sizes of the keys, since this was the reason why we couldn't simply use NCE. Since each function f takes x as input, the size of equivocable set is $2^{|x|}$, and by the property of NCE, the key size only depends on $|x|$, but not on the plaintext size (in fact, with our implementation of FEE the key size will be just $\lambda|x|$). Thus, even though each key has to equivocate *two* plaintexts of size $|k|$ each and would otherwise have to grow, with FEE the key size can be set to $\lambda|x|$ throughout the circuit.

Constructing FEE. We construct FEE using the Yao garbled circuits again — whereas this time it is the standard, statically secure version. In a nutshell, a real FEE key k is a simulated garbled input in the Yao garbled circuit, i.e. a set of n random labels for the input wires of a garbled circuit with n-bit input. A real FEE ciphertext encrypting message m with respect to the key k is a simulated garbled circuit (consistent with labels from k) with output m. Decryption works by evaluating the ciphertext (i.e. the simulated garbled circuit with output m) using the key (i.e. simulated input), which results in m.

A simulated FEE ciphertext for function f is a real garbling of f. A simulated FEE key, equivocating c for message m (such that $f(x) = m$ for some x), is a real garbled input x. Indeed, note that the real ciphertext and the key are indistinguishable from simulated by security of the garbling scheme; in particular, decrypting the simulated ciphertext (i.e. a real garbled f) with the simulated key (i.e. a real garbled x) results in computing $f(x) = m$.

In a bit more detail, recall that an FEE scheme is parameterized by s, n, l where 2^l is the size of the plaintext space, 2^n is the size of the set R of messages that a dummy ciphertext can open to, and s is the size of the description of the function $f : \{0,1\}^n \to R$. The scheme proceeds as follows. Let $U_{s,n,l}$ be the universal circuit that takes an s-bit description of a function f from n to l bits, and an n-bit value x, and outputs the l-bit value $f(x)$. The key k consists of n labels $k_1, ..., k_n$, where as usual each label is a random λ-bit string. To encrypt an l-bit message m, let I_m describe the constant function that outputs m on all inputs, and construct a simulated (using the static simulator) garbled evaluation of $U(I_m, 0^n)$, where the n labels that correspond to the input of I_m are the key $k_1, ..., k_n$. Specifically, the ciphertext consists of one label for each wire of $U_{s,n,l}$, except for the labels that correspond to the 0^n input. The FEE ciphertext also contains four ciphertexts per gate of $U_{s,n,l}$. One of these four ciphertexts is an encryption of the output label using the two input labels as the key, and the other three ciphertexts are just random strings. The ciphertexts are computed using a standard symmetric encryption scheme that is compatible with Yao static garbling.

Decryption amounts to evaluating the garbled circuit in the ciphertext using the labels $k_1, ..., k_n$ in the key for the input wires.

To generate a dummy ciphertext c_f for function f, prepare a real Yao static garbling of the circuit $U(f, \cdot)$, along with one label for each function wire and two labels for each input wire. c_f consists of the garbled gates and the labels for the function wire for $U(f, \cdot)$, and the trapdoor consists of the two labels for each input input wire for $U(f,\cdot)$.

To demonstrate a key k_x such that $Dec(k_x, c_f) = f(x)$, give the label for each input wire for $U(f,\cdot)$ that corresponds to input value x. To show randomness r_x such that $Enc(k_x, f(x), r_x) = c_f$, give the randomness used to encrypt the active ciphertext in each garbled gate in c_f. The other three ciphertexts in each garbled gate are presented as randomly chosen strings.

It can be seen that all the FEE properties are met. In particular, the tuple $m = f(x), k_x, r_x, c_f$ is indistinguishable from m, k, r, c where k, r are random and $c = Enc(k, m, r)$. (We note that above account is a bit of of an oversimplification of the definition and construction of FEE. See more details within.)

Two-party secure computation. Constructing two-party secure computation from equivocal garbling in secure channel setting (which can be implemented using any NCE), given adaptively secure OT (e.g. OT based on augmented NCE [CLOS02a]) is straightforward: First P_2 (the evaluator) sends the first message in n 1-out-of-2 OTs to P_1 (the garbler). Then P_1 generates the garbled circuit and sends it to P_2. P_1 then sends the second OT messages, where P_2's input to the ith OT is his ith input bit, and P_1's inputs to the ith OT are the two labels for the ith input wire for P_2. Finally P_2 evaluates the garbled circuit and announces the results.

Here we will describe the simulation for the harder case (where the adversary waits until the protocol ends, then corrupts the evaluator, and then the garbler). Since we assumed secure channels, the simulator doesn't need to do anything until the first corruption. When the evaluator is corrupted, the simulator learns x_2 and y, simulates the garbled input $\widetilde{x_1, x_2}$ (without even using x_2), and simulates the garbled circuit for output y. It also simulates OT messages (for OT output $\tilde{x}_2$). When the garbler is corrupted, the simulator learns x_1 (and therefore now it knows the whole input $x_1 x_2$), and uses the simulator of equivocal garbling to come up with randomness used to garble the circuit (in particular, both sets of keys). Next it uses OT simulator to simulate random coins of OT for inputs k_w^0, k_w^1 for each input wire of P_2.

The multiparty case. Our muliparty protocol is a variant of the BMR protocol [BMR90]. Recall that the idea of the BMR protocol is to have the parties jointly generate a single garbled circuit in such a way that all parties obtain all the garbled gates, and in addition each party P_i obtains one label for each input wire that's associated with itself. The label will correspond to P_i's input value for this wire. Then the parties broadcast their labels to each other (without the association between labels and values, of course). Finally each party locally evaluates the garbled circuit and obtains the output value. Since all gates can be garbled in parallel, the number of rounds of the protocol corresponds to the number of rounds needed to evaluate a single garbled gate. If generic MPC is used (e.g. [GMW87]) then the number of rounds is proportional to the security parameter. To get around that, BMR use the structure of Yao garbling to come up with a constant rounds protocol. Essentially, each party chooses labels and encrypts them locally, and the only computation that's joint is the xoring of the plaintexts, the additive sharing of the labels, and the choice of the random ordering of the four ciphertexts in each garbled gate. This simple computation can indeed be done in a constant number of rounds.

We keep this structure; however now we are faced with a number of additional challenges. First, since each of the four ciphertexts associated with a garbled gate now consists of multiple "individual ciphertexts", where each individual ciphertext is generated by a single party, it is not a priori clear how to make sure that the functions embedded in the dummy ciphertexts can be evaluated on the entire input to the computation. Indeed, it does not suffice that each of the ciphertexts has access to the input of only one party. Furthermore, since now all parties generate ciphertexts, the simulator now has to generate simulated encryption randomness whenever any party is corrupted — even before the entire input is known.

We get around these problems by designing different functions to be embedded in the dummy ciphertexts. In a nutshell, each party contributes one FEE ciphertext to each of the four ciphertexts for each gate. The function embedded in each dummy ciphertext has two modes: in the "random" mode, the function outputs a predetermined random value. In the "compute" mode, the function expects to get the full input to the circuit, and returns the appropriate output label xored with the all the predetermined random values. Later on, the simulator will make sure that the keys exhibited for all but the last party to be corrupted activate the random mode in their functions. The keys presented by the last party to be corrupted activate the "compute" mode of the embedded function. This way, overall, the labels of the simulated garbled circuits play the same role as in the two-party case.

The Byzantine case. We compile our BMR-style honest-but-curious multiparty protocol to a protocol that withstands Byzantine faults. This is done in two steps: First we obtain a constant-rounds protocol in the CRS model. This is done generically using the [CLOS02a] compiler. Note that this compiler preserves adaptive security while increasing the number of rounds by a factor of at most 3. Any hardness assumption that implies augmented NCE suffices. (In contrast, all existing constant-rounds adaptively secure multiparty computation protocols in the CRS model use indistinguishability obfuscation in an essential way.)

The second step replaces the CRS modeling with the constant-rounds adaptively secure coin tossing protocol of [GS12]. (Recall that while the overall protocol of [GS12] only obtains adaptive security for all-but-one corruptions, their underlying coin tossing protocol is indeed secure even if all parties are eventually corrupted.) We note that in order to be able to use the [GS12] protocol the CRS must be "public coins" - i.e. it should essentially be uniformly distributed. We can still use [CLOS02a] to do that, at the price of assuming dense cryptosystems. In addition, the [GS12] protocol uses collision resistant hash functions.

Obtaining leakage resilience. As discussed earlier in the introduction, we also construct a variant of the above multiparty protocol that's leakage tolerant as in [BCH12, BDL14]. It was shown in [BCH12] that any adaptively secure protocol where the simulation is *oblivious* is also leakage tolerant. Here oblivious simulation means that the simulated state of each corrupted party must be computed "locally", based only on the input and output of that party, plus perhaps some joint randomness that was sampled ahead of time and is available upon corruption of any party. In particular, the simulated state of a party cannot depend on the input or output of another party, even if that party is already corrupted.

Following [BDL14], we assume that the parties can first interact in a leak-free environment to sample some joint random state before the inputs are known. A first thought might be to simply run the above BMR-style protocol, where the initial sampling of the garbled circuit is done in the leak-free stage, and furthermore all randomness other than the output of that computation is erased. However note that the simulation in that protocol is inherently non-oblivious, since the simulator of the last party to be corrupted need to know the inputs of all parties.

We thus construct a new protocol that implements the BMR paradigm in a very different way. We start by extending the notion of FEE to the setting where each key and each ciphertext is generated jointly by a several participants. We call this new notion "Functional Equivocal Group Encryption (FEGE)". Now, rather than having each of the four ciphertexts in a garbled gate consist of multiple individual FEE ciphertexts, we let each one of these four ciphertexts be a single FEGE ciphertext that was generated by all the parties. This allows creating dummy ciphertexts that embed functions f that depend on values encoded in different pieces of the joint "group key". Now, when each party is corrupted, the simulator for that party will make sure that the simulated labels associated with that party will encode the input of that party. This way, as soon as all parties are corrupted, the overall input to f will encode the entire input to the circuit and the same functional equivocation mechanism will operate when the simulated garbled circuit is evaluated.

We can only make this idea work in the setting where the joint sampling of the labels takes place in a leak-free environment where all randomness except for the output of the preprocessing stage can be erased. Still, we obtain the first multiparty computation protocols in the plain model where all parties can be eventually corrupted, and the simulation is oblivious (modulo the initial offline sampling stage).

Public-key FEE. We extend FEE to the public-key setting. Similarly to symmetric FEE, a public-key FEE, or PK-FEE scheme is parameterized by the set $F_{s,n,l}$ of functions from n bits to l bits, whose description size is at most s bits. (Again, we consider the case where $n << l$.) Key generation, encryption and description are standard, where the size of the public and private keys is $O_\lambda(n)$ and the size of ciphertexts is $O_\lambda(s)$.

The algorithms for generating dummy ciphertexts and dummy encryption randomness have similar functionality as in the symmetric case. The algorithm for generating dummy secret keys is now naturally extended to generate not only a dummy decryption key; rather it should generate dummy randomness for the key generation algorithm, that will be consistent with the existing public encryption key and the dummy decryption key.

PK-FEE can be used to shorten the size of keys beyond what is possible in the case of standard non-committing encryption (NCE), while preserving the ability to generate dummy ciphertexts that can be opened to messages of choice. The ability to choose f and x separately provides additional flexibility. For instance, if f is a pseudorandom generator from n to l bits, then the scheme can be used to encrypt arbitrary l-bit messages, and then to generate dummy ciphertexts that later open to random-looking l-bit messages, and get away with keys of size $O_\lambda(n)$.

We construct PK-FEE from symmetric FEE and NCE, as follows. Key generation generates a keypair (k_e, k_d) for an NCE scheme for encrypting a key for the symmetric FEE scheme, namely $O_\lambda(n)$ bits. To encrypt a message $m \in \{0,1\}^l$, choose an FEE key k, compute $c_1 = FEE.Enc(k, m)$, $c_2 = NCE.Enc(k_e, k)$, and let $c = c_1, c_2$. Decryption is done accordingly.

To generate a dummy ciphertext for function f run $FEE.\mathsf{SimEnc}(f)$ to obtain c_f and $NCE.Sim$ to obtain a dummy ciphertext $\tilde{c}$, and output $c_f, \tilde{c}$, along with the state information from both simulators as trapdoor.

To generate a dummy key k_x for value $x \in \{0,1\}^n$, first obtain $k_x = FEE.Equiv(x, td)$, where td is the trapdoor generated by $FEE.\mathsf{SimEnc}$. Next use $NCE.Sim$ again to obtain the key generation randomness that leads to decryption of $\tilde{c}$ to k_x.

Finally, to generate encryption randomness, run $FEE.\mathsf{Adapt}(x, k_x, td)$ and $NCE.Sim$ again. The definition of security and the analysis follow naturally.

1.3 Related notions

Some definitions, mentioned in the introduction and related to our work, have very subtle differences between them, and similar names. In this section we comment on the differences.

Definitions related to FEE. Recall that the reason for introducing FEE was to shorten the NCE secret key, by limiting equivocation. We remark that *somewhere equivocal encryption* [HJO+16] and *somewhat non-committing encryption* [GWZ09] were introduced to achieve very similar goals (although not in the context of equivocal garbling). Despite the fact that our definition of FEE is similar in spirit, it is in fact a stronger primitive. We underline that previous definitions do not seem to suffice for our construction of equivocal garbling.

Roughly, *somewhere equivocal encryption* allows to encrypt l bits such that later bits in n predetermined positions, but not other bits, can be equivocated; the secret key has to be proportional to n, but not l (where n is significantly smaller than l). This is a special case of FEE for 2^n-sized set R with fixed $l - n$ bits. However, our construction of equivocal garbling requires equivocating to more complicated sets.

Somewhat non-committing encryption is closer to our FEE, since it allows to explain a dummy ciphertext for l-bit messages as any message from a predetermined set of size 2^n; however, the construction of [GWZ09] works only for polynomial-sized sets, since their communication is proportional to 2^n. In contrast, for our construction we need to be able to equivocate to sets of size $2^{|x|}$, which could be exponential. In addition, their construction is an interactive protocol, rather than a non-interactive encryption required in Yao garbled circuits.

Definitions of adaptive garbling. Note that our equivocal garbling is essentially an *adaptively secure* garbling, where "adaptive security" means security against adaptive corruptions. That is, the garbling should remain secure even if the adversary prefers to see the communication first (i.e. the garbled circuit and garbled input), and later corrupt the garbler and see its internal state (i.e. randomness used to garble the circuit).

However, we prefer to use a different name - equivocal garbling - since the term "adaptive garbling" was used to denote a different security definition. *Adaptive garbling*, or *garbling with adaptive choice of inputs* [GKR08, BHR12], requires that the garbled circuit and input can be simulated even though the input is chosen *after* the adversary sees the garbled circuit. That is, they allow the adversary to first determine C and see garbled $\tilde{C}$, and only then determine x and see $\tilde{x}$. $\tilde{C}$ should be simulatable given only C, and $\tilde{x}$ should be then simulated given in addition $y = C(x)$, but not x.

This definition is incomparable to our definition of equivocal garbling: indeed, in adaptively-chosen-inputs garbling the simulator doesn't have to simulate random coins of the party. On the other hand, here x can be chosen adaptively by the adversary after seeing $\tilde{C}$, whereas in equivocal garbling x has to be fixed in advance. (Such selective choice of inputs suffices for two party computation, since in the Yao protocol both inputs of the parties are already fixed by the time the garbled circuit has to be generated, i.e. after the first message is sent).

2 Definitions

A garbling scheme. Intuitively, a garbling scheme takes a circuit C and an input x and generates their garbled versions $\widetilde{C}, \widetilde{x}$ such that:

- Given $\widetilde{C}, \widetilde{x}$, it is possible to compute $y = C(x)$;
- $\widetilde{C}, \widetilde{x}$ don't reveal anything about x except $y = C(x)$.

The latter requirement is formalized by requiring that the simulator, who only knows C and y, can simulate the garbled circuit $\widetilde{C}$ and the garbled input $\widetilde{x}$ without knowing x.

In this paper we consider definitions of garbling with additional property called *bit decomposability*, which states that each bit of input x can be garbled without knowing other bits. This property will be required both for constructing FEE from statically secure garbling and for constructing two-party computation from adaptively secure garbling.

Definition 1 (Statically Secure Garbling Scheme). *We say that* $(\mathsf{Gen}, \mathsf{Garble}_{\mathsf{Prog}}, \mathsf{Garble}_{\mathsf{Inp}}, \mathsf{Eval})$ *is a statically secure garbling scheme, if the following properties hold:*

- **Correctness:**

$$\Pr[r \leftarrow \{0,1\}^{|r|}; K \leftarrow \mathsf{Gen}(1^\lambda); \widetilde{C} \leftarrow \mathsf{Garble}_{\mathsf{Prog}}(K, C; r); \{\widetilde{x}_i\}_{i=1}^n \leftarrow \mathsf{Garble}_{\mathsf{Inp}}(K, x_i, i) : \\ \mathsf{Eval}(\widetilde{C}, \widetilde{x}) = C(x)] > 1 - \mathsf{negl}(\lambda);$$

- **Static security:** *There exists a PPT algorithm* Sim*, such that any PPT adversary* A *wins the following game with at most negligible advantage:*

1. $A(1^\lambda)$ gives a circuit C and an input x to the challenger;

2. The challenger flips a bit b.
If $b = 0$:
- *It chooses random key $K \leftarrow \mathsf{Gen}(1^\lambda)$ and randomness of garbling r;*
- *It sets $(\widetilde{C} \leftarrow \mathsf{Garble}_{\mathsf{Prog}}(K, C; r), \{\widetilde{x}_i\}_{i=1}^n \leftarrow \mathsf{Garble}_{\mathsf{Inp}}(K, x_i, i)$;*
- *It sends $\widetilde{C}, \widetilde{x}$ to the adversary.*

If $b = 1$:
- *It sets $y = C(x)$;*
- *It runs the simulator $(\widetilde{C}, \widetilde{x}) \leftarrow \mathsf{Sim}(1^\lambda, C, y)$*
- *It sends $\widetilde{C}, \widetilde{x}$ to the adversary.*

3. The adversary outputs a bit b'.

The adversary wins if $b = b'$.

We will require one additional property of the statically-secure garbling scheme which is *obliviousness*. Roughly speaking, this property requires that there be a mechanism to reveal an honestly computed garbling as one that was computed by the simulation. A standard garbling mechanism using the Yao's garbling scheme computed using an encryption scheme which has pseudorandom ciphertexts can be shown to have this obliviousness property. On a high level it will suffice for the simulator to generate the inactive garbled rows as random strings and have the real ciphertexts in the inactive rows of an honestly computed garbling revealed as random strings. We define this obliviousness property next.

Oblivious sampling. There exists a PPT algorithm oSamp such that for any polynomial-time circuit C and for all input output pairs (x, y) such that $C(x) = y$ it holds that the following two distributions are indistinguishable.

$$\{r \leftarrow \{0,1\}^{|r|}; K \leftarrow \mathsf{Gen}(1^\lambda); \widetilde{C} \leftarrow \mathsf{Garble}_{\mathsf{Prog}}(K, C; r); \widetilde{x} \leftarrow \mathsf{Garble}_{\mathsf{Inp}}(K, x) : (\mathsf{oSamp}(r, K, x), \widetilde{C}, \widetilde{x})\}$$
$$\{R \leftarrow \{0,1\}^{|R|}; (\widetilde{C}, \widetilde{x}) \leftarrow \mathsf{Sim}(1^\lambda) : (R, \widetilde{C}, \widetilde{x})\}$$

2.1 Equivocal Garbling Scheme

Definition 2 (Equivocal garbling scheme). *We say that* $(\mathsf{Garble}_{\mathsf{Prog}}, \mathsf{Garble}_{\mathsf{Inp}}, \mathsf{Eval})$ *is an equivocal (adaptively secure) garbling scheme, if the following properties hold:*

- **Correctness:**

$$\Pr[r \leftarrow \{0,1\}^{|r|}; K \leftarrow \mathsf{Gen}(1^\lambda); \widetilde{C} \leftarrow \mathsf{Garble}_{\mathsf{Prog}}(K, C; r); \{\widetilde{x}_i\}_{i=1}^n \leftarrow \mathsf{Garble}_{\mathsf{Inp}}(K, x_i, i) : \mathsf{Eval}(\widetilde{C}, \widetilde{x}) = C(x)] > 1 - \mathsf{negl}(\lambda);$$

- **Security:** *There exists a pair of PPT algorithm* $(\mathsf{Sim}_1, \mathsf{Sim}_2)$*, such that any PPT adversary A wins the following game with at most negligible advantage:*

  1. A gives a circuit C and an input x to the challenger;

  2. The challenger flips a bit b.
  If $b = 0$:
  - *It chooses random garbling key K and randomness r;*
  - *It sets $(\widetilde{C} \leftarrow \mathsf{Garble}_{\mathsf{Prog}}(K, C; r), \{\widetilde{x}_i\}_{i=1}^n \leftarrow \mathsf{Garble}_{\mathsf{Inp}}(K, x_i, i)$;*

  - *It sends $\widetilde{C}, \widetilde{x}, K, r$ to the adversary.*

  If $b = 1$:

  - *It sets $y = C(x)$;*
  - *It runs the simulator $(\widetilde{C}, \widetilde{x}, \mathsf{state}) \leftarrow \mathsf{Sim}_1(C, y)$*
  - *It runs the simulator $(K, r) \leftarrow \mathsf{Sim}_2(\mathsf{state}, x)$*
  - *It sends $\widetilde{C}, \widetilde{x}, K, r$ to the adversary.*

3. *The adversary outputs a bit b'.*

The adversary wins if $b = b'$.

As explained before (see section 1.3), we reiterate here that our notion of equivocal garbling is different from the notion of adaptive garbling that requires security against an adaptive choice of inputs [GKR08, BHR12].

2.2 Convention for Garbling Schemes

We rely heavily on the Yao garbling scheme. We will follow some conventions when we describe garbling schemes.

- λ denotes the security parameter.

- We will use C to denote a circuit with description size $|C|$.

- Suppose C has an n-bit input and 1-bit output. The wires are numbered from 1 through m where we assume that wires are from 1 to n are input wires, and wire m is an output wire. We use κ to denote the size of keys k_w^0, k_w^1 for each wire w.

- We denote by bit_w the bit assigned to wire w by the computation $C(x)$.

- Typically, when we discuss a particular gate g, we use the notation α, β to denote the wire numbers of the input wires of the gate and γ the wire number for the output wire of the same gate.

- **Induction.** We will carry out induction by demonstrating as base case that the property holds for the input wires of the circuit. Then in an induction step we show that if for any gate the property holds for the input wires of a gate then it holds for the output wire of that gate. Then by considering a standard topological ordering of the gates the property will hold for output wires by the principle of mathematical induction.

  A common property (invariant) that we will repeatedly use in this work is the following: for every wire there will be three bits associated, bit_w that will represent the "actual" value flowing through that wire, λ_w a HIDDEN MASKS and Λ_w the VISIBLE MASKS. We will maintain the invariant that $\mathsf{bit}_w = \lambda_w \oplus \Lambda_w$. These masks have been so labelled to reflect what is visible and hidden from the evaluator of the garbled circuit.

- A garbled circuit comprises of garbled gates and each garbled gate comprises of 4 garbled rows. The garbled rows will be permuted according to the visible masks.

- We call keys $k_w^{\mathsf{bit}_w}$ (participating in the computation $C(x)$) **active keys**, and the other key for wire w, namely, $k_w^{1 \oplus \mathsf{bit}_w}$ as **inactive keys**. Analogously, the garbled row that corresponds to two active keys will be called the **active rows** and the remaining three rows as **inactive rows.**

	Left Column	Right Column
Key	$k_\alpha^{b\oplus\lambda_\alpha}$	$k_\beta^{b'\oplus\lambda_\beta}$
Ciphertext	$c_{g,\mathsf{left}}^{bb'}$	$c_{g,\mathsf{right}}^{bb'}$
Share	$s_{g,\mathsf{left}}^{bb'}$	$s_{g,\mathsf{right}}^{bb'}$

Table 1: Quick reference for convention for contents of Row (b, b') in Gate g.

- **Point-and-permute.** For a gate with input wires α, β and output wire γ, in row (b, b') for $b, b' \in \{0, 1\}$, we will use keys corresponding $b \oplus \lambda_\alpha$ and $b' \oplus \lambda_\beta$ and we will encrypt the key corresponding to $v = g(b \oplus \lambda_\alpha, b' \oplus \lambda_\beta)$ and mask $v \oplus \lambda_\gamma$. According to this convention, it is easy to see that the active row determined by bit_α and bit_β is $(\Lambda_\alpha = \mathsf{bit}_\alpha \oplus \lambda_\alpha, \Lambda_\beta = \mathsf{bit}_\beta \oplus \lambda_\beta)$ and the value encrypted will be $g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta)$.

A note on how the garbled rows are computed. Typically in a Yao garbling scheme the key (or message) to be encrypted in a row will be "double encrypted", i.e. first encrypted with the key for wire β and next with key for wire α. We use a different double encryption: we will apply XOR-based secret sharing to the key: $k_\gamma = s_{\mathsf{left}} \oplus s_{\mathsf{right}}$, and encrypt one share with the one key and the other share with the other key, thus obtaining a pair of ciphertexts $c_{\mathsf{left}}, c_{\mathsf{right}}$. Table 1 describes how the shares and ciphertexts will be denoted for a gate g with input wires α, β and output wire γ in Row (b, b'). Without loss of generality α will be the left wire and β will be the right wire.

Gate g looks like:

$$G_g = \begin{cases} (c_{g,\mathsf{left}}^{00}, & c_{g,\mathsf{right}}^{00}), \\ (c_{g,\mathsf{left}}^{01}, & c_{g,\mathsf{right}}^{01}), \\ (c_{g,\mathsf{left}}^{10}, & c_{g,\mathsf{right}}^{10}), \\ (c_{g,\mathsf{left}}^{11}, & c_{g,\mathsf{right}}^{11}) \end{cases}$$

3 Functionally Equivocal Encryption

In this section we define and construct the symmetric-key encryption called *functionally equivocal encryption* (FEE). This encryption is adaptively secure, meaning that the simulator can generate a dummy ciphertext (without knowing the plaintext $m \in M$) and later equivocate it to some plaintext m': that is, it can show encryption randomness r_{Enc} and the key k which are consistent with plaintext m' and the simulated ciphertext.

What makes FEE different from a non-committing encryption is that the equivocation is limited: the simulator cannot equivocate to *any* $m' \in M$, it can only equivocate to messages in the range of some function f (f has to be determined at the moment when the simulated ciphertext is generated). That is, the simulated ciphertexts (generated with respect to a function f) can be equivocated to a message m' only if $m' = f(x)$ for some x; x can be thought of as a "a short description" of m' with respect to a function f. Another important difference is that FEE keys are *succinct*: that is, the size of the key only depends on the size of description x (which we call *equivocation parameter*) , and not the size of the plaintext, which could be much larger.

3.1 Definitions

We present two definitions of FEE. In the first, basic, definition we present FEE as a natural generalization of non-committing encryption. Our second definition is a "garbling-friendly" definition, where we do several syntactic changes and consider security with respect to several functions. We will be using the second definition in our main construction of equivocal garbling.

Basic definition. We first describe the syntax:

- **Key generation.** $\mathsf{Gen}(1^\lambda, 1^n; r_{\mathsf{Gen}})$ takes as input security parameter λ, equivocation parameter n, and randomness of size $\mathsf{poly}(\lambda, n)$. It sets the key $k = r_{\mathsf{Gen}}$ and outputs it.

  Note that the key size only depends on equivocation parameter and security parameter, but not on the plaintext size.

- **Encryption.** $\mathsf{Enc}_k(f, m; r_{\mathsf{Enc}})$ outputs an encryption of m using randomness r_{Enc} and key k.

- **Decryption.** $\mathsf{Dec}_k(c)$ decrypts ciphertext c using key k and outputs plaintext m.

- **Ciphertext simulation.** $\mathsf{Sim}_1(f, r_{\mathsf{Sim}})$ uses its random coins to generate a simulated ciphertext c_{eq} together with a trapdoor td which can later be used for equivocation of c_{eq} to any $f(x)$.

- **Equivocation.** $\mathsf{Sim}_2(\mathsf{td}, f, x, c_{\mathsf{eq}})$ uses the equivocation trapdoor td and a short description x of plaintext $f(x)$ to generate a key k_{eq} and randomness r_{eq} consistent with ciphertext c_{eq} and plaintext $f(x)$; that is, $\mathsf{Enc}_{k_{\mathsf{eq}}}(\mathsf{params}(f), f(x); r_{\mathsf{eq}}) = c_{\mathsf{eq}}$ and $\mathsf{Dec}_{k_{\mathsf{eq}}}(c_{\mathsf{eq}}) = f(x)$.

Definition 3. **(Functionally equivocal encryption: basic definition.)**
A tuple of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Sim}_1, \mathsf{Sim}_2)$ *is a functionally equivocal encryption for a message space* $M = \{0,1\}^l$ *with equivocation parameter* n*, if the following properties hold:*

- **Correctness.** *For any* $m \in M$ $\Pr[\mathsf{Dec}_k(\mathsf{Enc}_k(f, m; r)) = m : r \leftarrow \{0,1\}^{|r|}, k \leftarrow \mathsf{Gen}(1^\lambda, 1^n)] > 1 - \mathsf{negl}(\lambda)$.

- **Security.** *For every* PPT *adversary* $\mathcal{A}$ *on input* 1^λ*, there exists a negligible function* $\nu(\cdot)$ *such that the probability that it wins the following game with challenger* $\mathcal{C}(1^\lambda)$ *is at most* $\frac{1}{2} + \nu(\lambda)$.

  1. *The adversary* $\mathcal{A}$ *sends a circuit* $f : \{0,1\}^n \to M$ *and an input* $x \in \{0,1\}^n$ *to* $\mathcal{C}$*;*
  2. $\mathcal{C}$ *computes the plaintext* $m \leftarrow f(x)$ *and chooses a bit* b *at random.*
  3. *If* $b = 0$, $\mathcal{C}$ *generates the real distribution:*
     - $\mathcal{C}$ *samples random FEE key* k *using* $\mathsf{Gen}(1^\lambda, 1^n)$ *and picks encryption randomness* r_{Enc}.
     - $\mathcal{C}$ *sets* $c_i \leftarrow \mathsf{Enc}_k(f, m; r_{\mathsf{Enc}})$*, where* $\mathsf{params}(f) = (|f|, n, l)$.
     - $\mathcal{C}$ *sends* (k, r_{Enc}, c) *to the adversary* $\mathcal{A}$.
  4. *If* $b = 1$, $\mathcal{C}$ *generates the simulated distribution:*
     - $\mathcal{C}$ *simulates the ciphertext* $(c_{\mathsf{eq}}, \mathsf{td}) \leftarrow \mathsf{Sim}_1(f, r_{\mathsf{Sim}})$ *using random* r_{Sim}*;*
     - $\mathcal{C}$ *equivocates* $(k_{\mathsf{eq}}, r_{\mathsf{eq}}) \leftarrow \mathsf{Sim}_2(\mathsf{td}, f, x, c_{\mathsf{eq}})$*;*
     - $\mathcal{C}$ *sends* $(k_{\mathsf{eq}}, r_{\mathsf{eq}}, c_{\mathsf{eq}})$ *to the adversary* $\mathcal{A}$.
  5. $\mathcal{A}$ *outputs a bit* b' *and wins if* $b = b'$.

- **Succinctness:** *The size of the key is polynomial in* λ, n *and independent of* l.

Garbling-friendly definition. Our construction of equivocal garbling will require a slightly modified definition of FEE, which we present here. The main differences are:

1. We split Sim_1 into two separate algorithms $\mathsf{SimTrap}$ (which generates the trapdoor) and SimEnc (which simulates a ciphertext). We also split Sim_2 into two separate algorithms Equiv (which equivocates the key) and Adapt (which equivocates randomness of encryption).
2. We consider security with respect to multiple ciphertexts (possibly corresponding to different functions $f_1, \ldots, f_t$); that is, the simulator should present a single equivocated key k_{eq} which decrypts multiple ciphertexts $c_1, \ldots, c_t$ to $f_1(x), \ldots, f_n(x)$ for a single description x.
3. We require that the encryption algorithm doesn't need to know the function f, and only needs to know its parameters $\mathsf{params}(f)$, which consist of description size $|f|$, its input length and its output length.

We describe the syntax:

- **Key generation.** $\mathsf{Gen}(1^\lambda, 1^n; r_{\mathsf{Gen}})$ takes as input security parameter λ, equivocation parameter n, and randomness of size $\mathsf{poly}(\lambda, n)$. It sets the key $k = r_{\mathsf{Gen}}$ and outputs it.

  Note that the key size only depends on equivocation parameter and security parameter, but not on the plaintext size.
- **Encryption.** $\mathsf{Enc}_k(\mathsf{params}, m; r_{\mathsf{Enc}})$ interprets params as function description size $|f|$, input length n and output length l. It outputs an encryption of m with respect to parameters params using randomness r_{Enc} and key k.
- **Decryption.** $\mathsf{Dec}_k(c)$ decrypts ciphertext c using key k and outputs plaintext m.
- **Ciphertext simulation.** Simulating a ciphertext comprises of two algorithms $(\mathsf{SimTrap}, \mathsf{SimEnc})$ where $\mathsf{SimTrap}$ on input $(1^\lambda, 1^n; r_{\mathsf{td}})$ outputs the trapdoor td and SimEnc on input $(f, \mathsf{td}; r_{\mathsf{Sim}})$ outputs a ciphertext c with respect to a function f.
- **Equivocation.** $\mathsf{Equiv}(x, \mathsf{td})$ uses the equivocation trapdoor td to generate a single fake key k_{eq} so that each simulated ciphertext $c_{\mathsf{eq},i}$, which was generated with respect to some function f_i and trapdoor td, decrypts to $f_i(x_i)$ under k_{eq}.
- **Randomness sampling.** $\mathsf{Adapt}(f, \mathsf{td}, r_{\mathsf{Sim}}, x)$ generates randomness r_{eq}, such that $\mathsf{Enc}_{k_{\mathsf{eq}}}(\mathsf{params}(f), f(x); r_{\mathsf{eq}}) = c$, where $c = \mathsf{SimEnc}(f, \mathsf{td}; r_{\mathsf{Sim}})$.

Definition 4. (Garbling-friendly FEE). *A tuple of algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{SimTrap}, \mathsf{SimEnc}, \mathsf{Equiv}, \mathsf{Adapt})$ *is a garbling-friendly functionally equivocal encryption, if the following properties hold:*

- **Correctness.** *For any* $m \in M$ $\Pr[\mathsf{Dec}_k(\mathsf{Enc}_k(f, m; r)) = m : r \leftarrow \{0,1\}^{|r|}, k \leftarrow \mathsf{Gen}(1^\lambda, 1^n)] > 1 - \mathsf{negl}(\lambda)$.
- **t-functional security.** *For every* PPT *adversary* $\mathcal{A}$ *on input* 1^λ*, there exists a negligible function* ν *such that the probability that it wins the following game with challenger* $\mathcal{C}(1^\lambda)$ *is at most* $\frac{1}{2} + \nu(\lambda)$.
  1. *The adversary* $\mathcal{A}$ *sends* t *functions* $f_1, \ldots, f_t$ *(where each* f_i *maps* n *bits to* l_i *bits) and an input* $x \in \{0,1\}^n$ *to* $\mathcal{C}$*;*
  2. $\mathcal{C}$ *computes the messages* $\{m_i \leftarrow f_i(x)\}_{i=1,\ldots,t}$.
  3. *Next it generates keys and ciphertexts in two different ways:*

- *$\mathcal{C}$ samples random FEE key k using* $\mathsf{Gen}(1^\lambda, 1^n)$ *and random strings* $r_{\mathsf{Enc},1}, \ldots, r_{\mathsf{Enc},n}$. *For* $1 \le i \le t$, *it computes*

$$c_i \leftarrow \mathsf{Enc}_k(\mathsf{params}_i, m_i; r_{\mathsf{Enc},i})$$

where $\mathsf{params}_i \leftarrow (|f_i|, n, l_i)$.

- *$\mathcal{C}$ computes ciphertexts using* SimEnc *as follows:*

$$\mathsf{td} \leftarrow \mathsf{SimTrap}(1^\lambda, 1^n; r_{\mathsf{td}}),$$
$$c_i \leftarrow \mathsf{SimEnc}(f_i, \mathsf{td}; r_{\mathsf{Sim},i}) \ \ \forall\, 1 \le i \ \le t$$

Next it computes $k_{\mathsf{eq}} \leftarrow \mathsf{Equiv}(x, \mathsf{td})$ *and*

$$r_{\mathsf{eq},i} \leftarrow \mathsf{Adapt}(f_i, \mathsf{td}, r_{\mathsf{Sim},i}, x) \ \ \forall\, 1 \le i \ \le t.$$

4. *$\mathcal{C}$ tosses a coin b.*
   - *If $b = 0$, $\mathcal{C}$ sends $(k, (c_1, r_{\mathsf{Enc},1}), \ldots, (c_t, r_{\mathsf{Enc},t}))$ to $\mathcal{A}$.*
   - *If $b = 1$, $\mathcal{C}$ sends $(k_{\mathsf{eq}}, (c_{\mathsf{eq},1}, r_{\mathsf{eq},1}), \ldots, (c_{\mathsf{eq},t}, r_{\mathsf{eq},t}))$ to $\mathcal{A}$.*
5. *A outputs a bit b'.*

A wins if $b = b'$.

3.2 Overview of FEE construction

We now construct a FEE scheme using *oblivious* version of Yao garbled circuits; oblivious means that a real garbled circuit can be claimed simulated by presenting convincing random coins of the simulation.

An overview. First we describe how to achieve the basic definition. On a high-level the construction works as follows:

The key k of the scheme with equivocation parameter n will be a simulated garbled input $\tilde{x}$, where $|x| = n$ (recall that simulated $\tilde{x}$ can be generated without knowing x). The encryption of m with respect to f under key $\tilde{x}$ will be a simulated garbled circuit $\tilde{f}$ (consistent with simulated garbled $\tilde{x}$), for output set to m. To decrypt a ciphertext ($\tilde{f}$) using the key ($\tilde{x}$), evaluate $\mathsf{Eval}(\tilde{f}, \tilde{x})$. Note that this evaluation results in m.

To simulate the ciphertext, the simulator generates the real garbled circuit $\tilde{f}$ and sets the trapdoor to be the garbling key. To equivocate FEE key k to plaintext $f(x)$, generate a real garbled input $\tilde{x}$ using the garbling key. Note that the simulated ciphertext decrypts to $f(x)$ under key $\tilde{x}$, since decryption runs evaluation of real garbled circuit $\tilde{f}$ on real garbled input $\tilde{x}$.

Finally, to simulate randomness of encryption, we use obliviousness of the Yao scheme and generate random coins of the simulation which are consistent with the real garbled circuit $\tilde{f}$.

Garbling-friendly FEE. Now we explain how to achieve garbling-friendly FEE. First note that the scheme can support multiple functions, since it is possible to generate a single garbled input x and many garbled circuits consistent with it.

To make encryption independent of description of f, we apply a universal transformation: namely, the key is still the simulated garbled input, but the ciphertext will be the garbled universal circuit together with the simulated garbled function $\tilde{f}$. The simulator can instead generate the ciphertext by creating real garbled circuit and garbling f honestly.

Finally, a closer look at the structure of Yao garbled circuits reveals that we can indeed achieve syntactic changes of garbling-friendly definition.

3.3 Construction of FEE

We now provide a formal description of the algorithms. Let $(\mathsf{Gen}^{\mathsf{CPA}}, \mathsf{Enc}^{\mathsf{CPA}}, \mathsf{Dec}^{\mathsf{CPA}})$ be a private-key encryption scheme with pseudorandom ciphertexts. Furthermore, we need the property that decrypting a random string with the key results in pseudorandom plaintext. We note that the PRF-based scheme $(r, F_k(r) \oplus m)$ satisfies this property.

Key Generation: Gen on input $(1^\lambda, 1^n; r_{\mathsf{Gen}})$ outputs n independent keys $k_1, \ldots, k_n$ sampled using $\mathsf{Gen}^{\mathsf{CPA}}(1^\lambda)$ along with n random bits $\Lambda_1, \ldots, \Lambda_n$.

Encryption: Enc with key $k = (k_1, \ldots, k_n)$ on input $(\mathsf{params} = (|f|, n, l), m; r_{\mathsf{Enc}})$, where $|m| = l$, generates ciphertext c as follows:

1. Let U be universal circuit which takes input of size n and function of size $|f|$ as input, and outputs an output of size $|m|$.
2. Consider an arbitrary topological order of the gates in U and label them in order $1, \ldots, s$.
3. Let W be the number of wires in the circuit that are given labels $1, \ldots, W$ so that all input wires are listed first and output wires are listed last. Sample a random bit Λ_w and a random key k_w using $\mathsf{Gen}^{\mathsf{CPA}}(1^\lambda)$ for every wire $n+1 \leq w \leq W$ that is not an output wire.
4. For $g = 1$ to s:

   Garble gate g: Let α, β be the input wire numbers and γ be the output of gate g. Let d be the length of ciphertexts when encrypting messages of length $|k_w|$ twice. Compute garbled gate $G_g = (R_g^{00}, R_g^{01}, R_g^{10}, R_g^{11})$ as follows: Let $c = \mathsf{Enc}^{\mathsf{CPA}}_{k_\alpha}(\mathsf{Enc}^{\mathsf{CPA}}_{k_\beta}(k_\gamma || \Lambda_c))$ if the output of the gate is not an output wire. If it is an output wire, let $c = \mathsf{Enc}^{\mathsf{CPA}}_{k_\alpha}(\mathsf{Enc}^{\mathsf{CPA}}_{k_\beta}(m_j))$ where the output wire corresponds to the j^{th} output bit of the function and $m = m_1 \cdots m_n$. For $0 \leq b, b' \leq 1$, set
   $$R_g^{bb'} = \begin{cases} c & \text{if } \Lambda_\alpha = b \text{ and } \Lambda_\beta = b' \\ r \leftarrow \{0,1\}^d & \text{o.w.} \end{cases}$$
5. Set simulated $\tilde{f}$ to be $(k_{n+1}, \ldots, k_{n+|f|})$ together with $(\Lambda_{n+1}, \ldots, \Lambda_{n+|f|})$.
6. Finally, set $c = (G_1, \ldots, G_s, \tilde{f})$.

Simulating ciphertexts. $\mathsf{SimTrap}(1^\lambda, 1^n; r_{\mathsf{td}})$ samples $2n$ keys using k_j^0 and k_j^1 for $1 \leq j \leq n$ using $\mathsf{Gen}^{\mathsf{CPA}}(1^\lambda)$ and $2n$ random bits $\lambda_1, \ldots, \lambda_n$ and outputs $\mathsf{td} = ((k_1^0, k_1^1, \ldots, k_n^0, k_n^1), (\lambda_1, \ldots, \lambda_n))$.

SimEnc on input $(f, \mathsf{td}; r_{\mathsf{Sim}})$, where $f : \{0,1\}^n \to \{0,1\}^{|m|}$, computes c_{eq} as follows:

1. Let U be universal circuit which takes input of size n and function of size $|f|$ as input, and outputs an output of size $|m|$.
2. Consider an arbitrary topological order of the gates in the U and label them in order $1, \ldots, s$ where s is the number of gates in U.
3. Let W be the number of wires in the circuit that are given labels $1, \ldots, W$ so that all input wires are listed first. Sample a random mask λ_w and a pairs of random keys k_w^0, k_w^1 using $\mathsf{Gen}^{\mathsf{CPA}}(1^\lambda)$ for every wire $n+1 \leq w \leq W$ that is not an output wire.
4. For $g = 1$ to s:

Garble gate g: Let α, β be the input wire numbers and γ be the output of gate g. Compute garbled gate $G_g = (R_g^{00}, R_g^{01}, R_g^{10}, R_g^{11})$ as follows:

$$\chi_1 = k_\gamma^v || (v \oplus \lambda_\gamma) \text{ where } v = g(\lambda_\alpha, \lambda_\beta) \qquad R_g^{00} = \mathsf{Enc}^{\mathsf{CPA}}_{k_\alpha^{\lambda_\alpha}}(\mathsf{Enc}^{\mathsf{CPA}}_{k_\beta^{\lambda_\beta}}(\chi_1))$$

$$\chi_2 = k_\gamma^v || (v \oplus \lambda_\gamma) \text{ where } v = g(1 \oplus \lambda_\alpha, \lambda_\beta) \qquad R_g^{01} = \mathsf{Enc}^{\mathsf{CPA}}_{k_\alpha^{1\oplus\lambda_\alpha}}(\mathsf{Enc}^{\mathsf{CPA}}_{k_\beta^{\lambda_\beta}}(\chi_2))$$

$$\chi_3 = k_\gamma^v || (v \oplus \lambda_\gamma) \text{ where } v = g(\lambda_\alpha, 1 \oplus \lambda_\beta) \qquad R_g^{10} = \mathsf{Enc}^{\mathsf{CPA}}_{k_\alpha^{\lambda_\alpha}}(\mathsf{Enc}^{\mathsf{CPA}}_{k_\beta^{1\oplus\lambda_\beta}}(\chi_3))$$

$$\chi_4 = k_\gamma^v || (v \oplus \lambda_\gamma) \text{ where } v = g(1 \oplus \lambda_\alpha, 1 \oplus \lambda_\beta) \qquad R_g^{11} = \mathsf{Enc}^{\mathsf{CPA}}_{k_\alpha^{1\oplus\lambda_\alpha}}(\mathsf{Enc}^{\mathsf{CPA}}_{k_\beta^{1\oplus\lambda_\beta}}(\chi_4))$$

Let $r_g^{bb'}$ be the randomness used to compute the encryptions in $R_g^{bb'}$ for $b, b' \in \{0,1\}$.

5. Set $\tilde{f}$ to be $(k_{n+1}^{f_1}, \ldots, k_{n+|f|}^{f_{|f|}})$ together with $(\lambda_{n+1} \oplus f_1, \ldots, \lambda_{n+|f|} \oplus f_{|f|})$.
6. Finally, set $c_{\mathsf{eq}} = (G_1, \ldots, G_s, \tilde{f})$.

Equivocation. Equiv on input (x, td) uses the trapdoor $\mathsf{td} = ((k_1^0, k_1^1, \ldots, k_n^0, k_n^1), (\lambda_1, \ldots, \lambda_n))$ to compute the key $k_{\mathsf{eq}} = ((k_1^{x_1}, \lambda_1 \oplus x_1), \ldots, (k_n^{x_n}, \lambda_n \oplus x_n))$ where $x = x_1 \cdots x_n$.

Randomness sampling. Adapt on input $(f, \mathsf{td}, r_{\mathsf{Sim}}, x)$ needs to generate a random string r_{eq} so that $\mathsf{Enc}_{k_{\mathsf{eq}}}(\mathsf{params} = \mathsf{params}(f), f(x); r_{\mathsf{eq}}) = c_{\mathsf{eq}}$. To generate r_{eq}, it proceeds as follows:

1. Let $c_{\mathsf{eq}} = (G_1, \ldots, G_s, \tilde{f})$ where each $G_g = (R_g^{00}, R_g^{01}, R_g^{10}, R_g^{11})$.
2. Reconstruct k_w^0, k_w^1, λ_w and the randomness used to generate each gate g from r_{Sim}. Compute the real value in each wire when the input is x. Let bit_w be the value in wire w. Set
$$\Lambda_w = \lambda_w \oplus \mathsf{bit}_w.$$
3. For each gate g, the randomness used for G_g is the concatenation of the randomness used for each row $R_g^{bb'}$ ($b, b' \in \{0,1\}$) which is computed as follows: Let α, β be the input wires and γ be the output wire. Let $r_g^{bb'}$ be the randomness used to compute $R_g^{bb'}$ by SimEnc.
   - For $b, b' \in \{0,1\}$, randomness for $R_g^{bb'}$: If $\Lambda_\alpha = b$ and $\Lambda_\beta = b'$, then set $\tilde{r}_g^{bb'} = r_g^{bb'}$. Otherwise set $\tilde{r}_g^{bb'} = R_g^{bb'}$.
4. set randomness r^f used to compute $\tilde{f}$ to be $k_{n+1}^{f_1}, \ldots, k_{n+|f|}^{f_{|f|}}$ and the masks $\Lambda_{n+1}, \ldots, \Lambda_{n+|f|}$.
5. Output $r_{\mathsf{eq}} = \{(\tilde{r}_g^{00}, \tilde{r}_g^{01}, \tilde{r}_g^{10}, \tilde{r}_g^{11})\}_{g \in [s]}$, together with r^f.

Theorem 1. (Gen, Enc, SimTrap, SimEnc, Equiv, Adapt) *described above is a garbling-friendly functionally equivocal encryption scheme.*

Proof. On a high-level, the proof of correctness and indistinguishability will follows from the standard properties of Yao garbling and the pseudorandomness of the ciphertexts computed using the underlying CPA secure encryption scheme. We provide a formal proof below.

Correctness. Correctness of the scheme follows from the fact that with overwhelming probability the simulated garbled circuit (for output m) on the simulated garbled input outputs m.

Security. Recall that the definition of a FEE requires to show that honestly generated encryptions and simulated encryptions along with the messages, random coins, and the key, are indistinguishable for t messages $f_1(x), \ldots, f_t(x)$ where $x \in \{0,1\}^n$ and f_i is a function from $\{0,1\}^n \to \{0,1\}^{\ell_i}$ for $1 \leq i \leq t$. More formally, we need to show that for any PPT adversary $\mathcal{A}$ the following two distributions are indistinguishable:

$$\begin{aligned}
D_0^n = \{(f_1, \ldots, f_t, x) &\leftarrow \mathcal{A}(1^\lambda, 1^n);\\
k &\leftarrow \mathsf{Gen}(1^\lambda, 1^n);\\
c_i &\leftarrow \mathsf{Enc}_k(\mathsf{params}_i, m_i; r_{\mathsf{Enc},i}) \text{ where } \mathsf{params}_i \leftarrow (|f_i|, n, l_i)\ \forall\, 1 \leq i \leq t :\\
&(k, (c_1, r_{\mathsf{Enc},1}), \ldots, (c_t, r_{\mathsf{Enc},t}))\}\\
D_1^n = \{(f_1, \ldots, f_t, x) &\leftarrow \mathcal{A}(1^\lambda, 1^n);\\
\mathsf{td} &\leftarrow \mathsf{SimTrap}(1^\lambda, 1^n);\\
c_{\mathsf{eq},i} &\leftarrow \mathsf{SimEnc}(f_i, \mathsf{td}, r_{\mathsf{Sim},i})\ \forall\, 1 \leq i \leq t;\\
k_{\mathsf{eq}} &\leftarrow \mathsf{Equiv}(\mathsf{td}, x)\\
r_{\mathsf{eq},i} &\leftarrow \mathsf{Adapt}(f_i, \mathsf{td}, r_{\mathsf{Sim},i}, x)\ \forall\, 1 \leq i \leq t;\\
&(k_{\mathsf{eq}}, (c_{\mathsf{eq},1}, r_{\mathsf{eq},1}), \ldots, (c_{\mathsf{eq},t}, r_{\mathsf{eq},t}))\}
\end{aligned}$$

We will consider a sequence of hybrids starting from the simulation to the real encryption.

Hybrid H_1^n. The output of this experiment is identical to D_1^n.

Hybrid $H_2^n[j](1 \leq j \leq n + |f_1| + \ldots + |f_t|)$. For all possible input wire of any universal circuit U_i (where U_i is a universal circuit for params_i), we consider the hybrid $H_2^n[j]$; it is identical to $H_2^n[j-1]$ except for that for every ciphertext $c_{\mathsf{eq},i}$ and every gate g that has j as one of its input wires, we make the following modification:

Let $G_g = (R_g^{00}, R_g^{01}, R_g^{10}, R_g^{11})$. Let $(\mathsf{bit}_w, \lambda_w, \Lambda_w)$ be the real value, hidden mask and visible mask computed for each wire $1 \leq w \leq W_i$ of U_i . Let $\alpha = j, \beta$ be the input wires and γ the output wire of gate g. Let $(r_g^{00}, r_g^{01}, r_g^{10}, r_g^{11})$ be the randomness reported as in hybrid H_1^n (i.e. real simulation). Let x_j be the j^{th} bit of the input x. Then $\Lambda_j = \lambda_j \oplus x_j$.

In $H_2^n[j]$, for every ciphertext c_i and every gate g that has j as its input wire, we replace G_g as follows:

$$c_{b'} = \mathsf{Enc}^{\mathsf{CPA}}_{k_j^{x_j}}(\mathsf{Enc}^{\mathsf{CPA}}_{k_\beta^{b' \oplus \lambda_\beta}}(k_\gamma^{\Lambda_\gamma} || \Lambda_\gamma)) \text{ with randomness } r_{\mathsf{Enc}};$$

$$(\widetilde{R}_g^{bb'}, \widetilde{r}_g^{bb'}) = \begin{cases} (c_{b'}, r_{\mathsf{Enc}}) & \text{if } b = \Lambda_j \oplus x_j \\ r \leftarrow \{0,1\}^d & \text{o.w.} \end{cases}$$

for each bit b'. By our definition $H_2^n[0] = H_1^n$. In Hybrid $H_2^n[j]$ for $1 \leq j \leq n + |f_1| + \ldots + |f_t|$ we replace all ciphertexts, that use inactive key for wire j as one of the encryption keys, to a random string. After this modification there will be no ciphertext encrypted under this key in any gate. Now, we can rely on the pseudorandomness property of encryptions under this key to argue indistinguishability of $H_2^n[j]$ and $H_2^n[j+1]$ for $0 \leq j < n + |f_1| + \ldots + |f_t|$. Therefore we have that

$$H_1^n \approx H_2^n[n + |f_1| + \ldots + |f_t|]$$

Hybrid $H_3^n[i, g]$. We consider a sequence of hybrids in the following order for every $1 \leq i \leq t$ and a gate g in U_i, which does not contain an input wire of the circuit (as they have been taken care of in the previous hybrid). Hybrid $H_2^n[i,g]$ is identical to $H_2^n[i,g-1]$ (if $g>1$ and identical to $H_2^n[i-1,s_{i-1}]$ if $g=0$) with the only exception that in $c_{\mathsf{eq},i}$, the part corresponding to gate g is modified as follows: Let $c_{\mathsf{eq},i}=(G_1,\ldots,G_s)$ and $G_g=(R_g^{00},R_g^{01},R_g^{10},R_g^{11})$. Let $(\mathsf{bit}_w^i,\lambda_w^i,\Lambda_w^i)$ be the real value, hidden mask and visible mask computed for each wire $1\le w\le W_i$ in U_i. Let α,β be the input wires and γ the output wire of gate g. Let $(r_g^{00},r_g^{01},r_g^{10},r_g^{11})$ be the randomness reported as in hybrid H_1^n. In $H_2^n[i,g]$ we replace the randomness and G_g in $c_{\mathsf{eq},i}$ as follows:

$$c=\mathsf{Enc}^{\mathsf{CPA}}_{k_\alpha^{\mathsf{bit}_\alpha^i}}(\mathsf{Enc}^{\mathsf{CPA}}_{k_\beta^{\mathsf{bit}_\beta^i}}(k_\gamma^{g(\mathsf{bit}_\alpha^i,\mathsf{bit}_\beta^i)}||\Lambda_\gamma))\text{ with randomness }r_{\mathsf{Enc}}$$

$$(\widetilde{R}_g^{bb'},\widetilde{r}_g^{bb'})=\begin{cases}(c,r_{\mathsf{Enc}}) & \text{if } b=\Lambda_\alpha \text{ and } b'=\Lambda_\beta\\ r\leftarrow\{0,1\}^\ell & \text{o.w.}\end{cases}$$

We will inductively show the following:

Claim 2. *For every i, we will show that $H_3^n[i,g]\approx H_3^n[i,g+1]$ for $0\le g<s_i$.*

Proof. Just as in the previous hybrid, indistinguishability follows from the pseudorandom ciphertext property of the underlying encryption scheme. Consider a gate g that contains no input wire from the Circuit that has input wires α,β and output γ. We will prove that for any such gate, in Hybrid $H_3^n[i,g-1]$, $k_\alpha^{1\oplus\Lambda_\alpha}$ and $k_\beta^{1\oplus\Lambda_\beta}$ have been removed from all ciphertexts. We will prove by induction that if the statement holds in Hybrid $H_3^n[i,g]$, the keys $k_\alpha^{1\oplus\mathsf{bit}_\alpha^i}$ and $k_\beta^{1\oplus\mathsf{bit}_\beta^i}$ have been removed from the ciphertexts, then for any gate g' that contains γ as in input wire it holds that in Hybrid $H_3^n(i,g')$ the key $k_\gamma^{1\oplus\mathsf{bit}_\gamma}$ would have been removed. We will show that we remove this key in the current hybrid and then the induction hypothesis follows as g' will always come after g in the topological order. On a high-level, only the "active keys" will survive and the rows encrypted using inactive keys will be set to random. This inductive step can be easily see from the modification we do in Hybrid $H_3^n[i,g]$. Namely, we replace three rows with a random string and one row remains the same. The one row that remains the same is encrypted using the keys $k_\alpha^{\mathsf{bit}_\alpha^i}$ and $k_\beta^{\mathsf{bit}_\beta^i}$. The remaining rows must be encrypted with at least one of the two keys $k_\alpha^{1\oplus\mathsf{bit}_\alpha^i}$ and $k_\beta^{1\oplus\mathsf{bit}_\beta^i}$. This completes the proof of the inductive step.

Given our induction hypothesis, arguing that $H_3^n[i,g]\approx H_3^n[i,g+1]$ are indistinguishable follows directly from the pseudorandomness property under the keys $k_\alpha^{1\oplus\mathsf{bit}_\alpha^i}$ and $k_\beta^{1\oplus\mathsf{bit}_\beta^i}$ (which have been removed). ☐

This concludes the proof of Theorem 1. ☐

4 From Functionally Equivocal Encryption to Eqivocal Garbling

In this section we describe our construction of equivocal garbling according to Definition 2, based on a functionally equivocal encryption (FEE), defined and built in Section 3. We start with an overview of the construction, and then proceed with the formal description and the proof.

4.1 An Overview

Conventions. Let C be a circuit with description size $|C|$ which takes as inputs n-bit strings. For simplicity, we present our construction for circuits that output a single bit; however, our construction can be naturally extended to multiple-bit output circuits. We denote by $m = n + \mathsf{gates}(C)$ the total number of "different" wires in C, i.e. if a gate has a fan-out more than 1, only one wire is counted. The wires are numbered from 1 through m where we assume that wires are from 1 to n are input wires, and wire m is an output wire. We use κ to denote the size of keys k_w^0, k_w^1 for each wire w. We also denote by bit_w the bit assigned to wire w by the computation $C(x)$. Typically, when we discuss a particular gate g, we use the notation α, β to denote the wire numbers of the input wires of the gate and γ the wire number for the output wire of the same gate.

Garbling. The garbling procedure will follow closely the standard Yao garbling scheme [Yao86] that includes the so called point-and-permute feature [MNPS04a], except that we use an FEE scheme instead of a CPA-secure encryption. Namely, the garbler first chooses a pair of FEE keys (k_w^0, k_w^1) for each wire w of the circuit C. (We assume that output wires of the same gate are labeled with the same index w and therefore are assigned the same pair of keys.)

Next the garbler garbles each gate using double encryption. Our double encryption of message m will consist of an encryption of shares of the message m, s_{left} and s_{right} such that $s_{\mathsf{left}} \oplus s_{\mathsf{right}} = m$ where s_{left} will be encrypted under the key of the left wire entering the gate and s_{right} using the right key. This guarantees that with both left and right keys it is possible to reconstruct m, but having only one key doesn't reveal any information about m.

More precisely, the garbler generates the following 4 pairs of ciphertexts for each gate g:

$$c_{g,\mathsf{left}}^{00} = \mathsf{FEE.Enc}_{k_\alpha^0}(s_{g,\mathsf{left}}^{00}), \quad c_{g,\mathsf{right}}^{00} = \mathsf{FEE.Enc}_{k_\beta^0}(s_{g,\mathsf{right}}^{00}),$$

$$c_{g,\mathsf{left}}^{01} = \mathsf{FEE.Enc}_{k_\alpha^0}(s_{g,\mathsf{left}}^{01}), \quad c_{g,\mathsf{right}}^{01} = \mathsf{FEE.Enc}_{k_\beta^1}(s_{g,\mathsf{right}}^{01}),$$

$$c_{g,\mathsf{left}}^{10} = \mathsf{FEE.Enc}_{k_\alpha^1}(s_{g,\mathsf{left}}^{10}), \quad c_{g,\mathsf{right}}^{10} = \mathsf{FEE.Enc}_{k_\beta^0}(s_{g,\mathsf{right}}^{10}),$$

$$c_{g,\mathsf{left}}^{11} = \mathsf{FEE.Enc}_{k_\alpha^1}(s_{g,\mathsf{left}}^{00}), \quad c_{g,\mathsf{right}}^{11} = \mathsf{FEE.Enc}_{k_\beta^1}(s_{g,\mathsf{right}}^{11}),$$

where α, β are input wires and γ are output wires of the gate and $s_{g,\mathsf{left}}^{bb'}$ and $s_{g,\mathsf{right}}^{bb'}$ are random XOR sharings of $k_\gamma^{g(b,b')}$. Ciphertexts for output gates encrypt output bits (padded to the size of the key) instead of the key. Note that FEE is a randomized encryption; the garbler encrypts each of 8 ciphertexts under freshly chosen randomness, which we omitted here for brevity.

As we said, the difference from the Yao garbled circuits is that Enc is an FEE scheme, which means that encryption algorithm should also take as input parameters params of a function which image can later be equivocated to. The garbler sets params to be $(|F|, n, \kappa)$, where $|F|$ is the size of functions F described on figure 1. This guarantees that the simulator will be able to generate dummy ciphertexts for plaintext size κ and later open them only to messages of the form $f(x)$, where $|f| = |F|$, $|f(x)| = \kappa$ and $|x| = n$. Such limited equivocation is, on one hand, enough to show adaptive security, and on the other hand, it guarantees that keys are not growing with the depth of the circuit. In particular the keys are proportional to $|x|$.

Each garbled gate G_g consists of all 4 pairs of ciphertexts as described above, shuffled in random order.

The garbled circuit consists of garbled gates $\{G_g\}_{g \in [\mathsf{gates}(C)]}$. The garbled input consists of keys corresponding to input bits, i.e. $(k_1^{x_1}, \ldots, k_n^{x_n})$.

Evaluation of the garbled circuit. The evaluator evaluates the circuit gate by gate, starting from input gates. Assume the evaluator already learned keys $k_\alpha^{\mathsf{bit}_\alpha}, k_\beta^{\mathsf{bit}_\beta}$, assigned to input wires α, β of gate g by $C(x)$. The evaluator tries to decrypt each double encryption of this gate, using this pair of keys and only one double encryption will decrypt successfully.[3] It gets $s_{\mathsf{left}} \leftarrow \mathsf{FEE.Dec}_{k_\alpha^{\mathsf{bit}_\alpha}}(c_{\mathsf{left}})$ and $s_{\mathsf{right}} \leftarrow \mathsf{FEE.Dec}_{k_\beta^{\mathsf{bit}_\beta}}(c_{\mathsf{right}})$ and learns two shares $s_{\mathsf{left}}, s_{\mathsf{right}}$ of the next key. It reconstructs the key as $k_\gamma^{g(\mathsf{bit}_\alpha,\mathsf{bit}_\beta)} = s_{\mathsf{left}} \oplus s_{\mathsf{right}}$. The evaluator continues the process, until it learns the output of the computation after decrypting the last gate.

Simulation. The simulation can be described in two parts: simulating the Garbled Circuit and simulating the internal randomness.

Simulating the Garbled Circuit. The simulator starts with preparing active keys, i.e keys corresponding to the execution $C(x)$: for this it chooses a random key k_w for each wire w of C. Also for each wire it generates an FEE trapdoor $\mathsf{td}_w \leftarrow \mathsf{SimTrap}$, which later will be used to equivocate inactive keys of all wires. For each gate the simulator randomly chooses which one of 4 rows will be an active row (i.e. for which pair of ciphertexts the evaluator will know both keys), by choosing random bits Λ_w for each wire w: for each gate, the active row will be row number $(\Lambda_\alpha, \Lambda_\beta)$, where α, β are input wires of that gate.

The simulator encrypts active pair of ciphertexts like in the real world, i.e. it secret shares k_g and sets

$$c_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta} = \mathsf{FEE.Enc}_{k_\alpha}(s_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta}), \;\; c_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta} = \mathsf{FEE.Enc}_{k_\beta}(s_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta}).$$

Another row contains the double encryption $c_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}, c_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}$ that should be encrypted under the key k_α selected for the active row, but under a different second key $\widehat{k}_\beta$. Furthermore, each encrypt shares that will add up to the intended key on wire γ. However, note that at the time of creating the ciphertexts, the simulator does not know what key to encrypt in this row. Since the key is secret shared it can of course encrypt a random share $s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}$ under the known key k_α to generate the left ciphertext $c_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}$. The right ciphertext is simulated using FEE simulator (to be later opened in such a way that two shares xor to the correct key).

More precisely, we will generate the right ciphertext $c_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}$ using SimEnc with respect to a particular function that will later allow us to define both the key $\widehat{k}_\beta$ and the right message this second ciphertext needs to be revealed as. In fact, the right ciphertext should encrypt masked key $s_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} = k \oplus s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}$, but at this moment the simulator doesn't know whether the key should be active k_γ or inactive $\widehat{k}_\gamma$ as it will depend on the real bit assignment $\mathsf{bit}_\alpha, \mathsf{bit}_\beta$ on wires α, β when evaluated on the real input x. Therefore, to generate the ciphertexts in the row $\Lambda_\alpha, 1 \oplus \Lambda_b$, the simulator does the following:

$$c_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} = \mathsf{FEE.Enc}_{k_\alpha}(s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}), \;\; c_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} = \mathsf{FEE.SimEnc}(F_g^{\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}], \mathsf{td}_\beta),$$

where $s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}$ is a random string and $F^{\Lambda_\alpha,1\oplus\Lambda_b}$ is a specially crafted function that we describe in Figure 1. The third row of ciphertexts is generated similar to the second, except that in this row the key for wire β is the active key k_β which is known and the key for the other wire α and the message is unknown. Therefore, this row will be:

[3]For the purpose of this overview, we assume that decrypting double ciphertext with the wrong pair of keys results in a detectable failure. This can be achieved by appending 0^λ string to the plaintext during encryption, and by verifying that the last λ bits of the decrypted plaintext is 0^λ during decryption. In our full construction however we don't need this assumption on the encryption scheme. We instead use a "point-and-permute" technique which tells the evaluator which one of 4 double encryptions it needs to decrypt.

$$c_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta} = \mathsf{FEE.SimEnc}(F_g^{1\oplus\Lambda_\alpha,\Lambda_\beta}[s_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}], \mathsf{td}_\alpha), \;\; c_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta} = \mathsf{FEE.Enc}_{k_\beta}(s_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}).$$

where $s_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}$ is a random string and $F^{1\oplus\Lambda_\alpha,\Lambda_b}$ is described in Figure 1.

Simulating the last pair is slightly different, since it is encrypted under the inactive keys $\widehat{k}_\alpha, \widehat{k}_\beta$, neither of which are determined yet by the simulator. Thus, the simulator generates both ciphertexts using FEE simulator. It simulates the first ciphertext with respect to a constant function that always outputs a share $s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}$, and the second with respect to a third function $F_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}$ as follows:

$$\begin{aligned} c_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} &= \mathsf{FEE.SimEnc}(\mathsf{Const}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}], \mathsf{td}_\alpha), \\ c_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} &= \mathsf{FEE.SimEnc}(F_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}], \mathsf{td}_\beta). \end{aligned}$$

where the function $\mathsf{Const}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}] \equiv s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}$ and function $F^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}[\mathsf{mask}]$ is the third specially crafted function.

Finally, the gate looks like as follows:

Row number	Left ciphertext FEE.	Right ciphertext FEE.
$(\Lambda_\alpha, \Lambda_\beta)$	$\mathsf{Enc}_{k_\alpha}(s_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta})$	$\mathsf{Enc}_{k_\beta}(s_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta})$
$(\Lambda_\alpha, 1\oplus\Lambda_\beta)$	$\mathsf{Enc}_{k_\alpha}(s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta})$	$\mathsf{SimEnc}(F_g^{\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}])$
$(1\oplus\Lambda_\alpha, \Lambda_\beta)$	$\mathsf{SimEnc}(F_g^{1\oplus\Lambda_\alpha,\Lambda_\beta}[s_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}])$	$\mathsf{Enc}_{k_\beta}(s_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta})$
$(1\oplus\Lambda_\alpha, 1\oplus\Lambda_\beta)$	$\mathsf{SimEnc}(\mathsf{Const}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}])$	$\mathsf{SimEnc}(F_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}])$

Table 2: Garbled gate g generated by the simulator.

The ciphertexts for an output gate are generated in a similar way, with the difference that the active double encryption encrypts the output $y = C(x)$ instead of the key. In essence, these functions will be the same as F, except that they output masked output bits instead of masked keys for the next gate (See Figure 1).

The simulator reorders 4 ciphertexts in each gate according to $(\Lambda_\alpha, \Lambda_\beta)$ and outputs them. Next it outputs the active keys for the input wires $(k_1, \ldots, k_n)$ as a garbled input.

Simulation of the internal state of the garbler. When the simulator is given input x, it needs to present inactive key $\widehat{k}_w$ for each wire w. It generates these keys by running $\widehat{k}_w \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_w; x)$ for each gate w.

Intuitively, this is indistinguishable from real garbling, since each simulated garbled gate looks like it was a real garbled gate generated using keys $k_\alpha, \widehat{k}_\alpha, k_\beta, \widehat{k}_\beta$, where k_α, k_β are active for the computation $C(x)$: Recall that the definition of FEE guarantees that the key $\widehat{k}_w$ generated as $\mathsf{FEE.Equiv}(\mathsf{td}_w; x)$ decrypts the ciphertext $c = \mathsf{FEE.SimEnc}(F)$ to $F(x)$. This means that decrypting, say, $c_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}$ with $\widehat{k}_\beta$ results in $F_g^{\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}](x)$, which is equal to the share that is either $k_\gamma \oplus s_{\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}$ or $\widehat{k}_\gamma \oplus s_{\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}$, depending on the assignment $\mathsf{bit}_\alpha, \mathsf{bit}_\beta$. Thus the second double encryption will decrypt to the correct key (k_γ or $\widehat{k}_\gamma$) under $k_\alpha, \widehat{k}_\beta$.

This in particular means that each garbled gate looks like a real garbled gate with active keys k_γ; for instance, if the gate was an AND gate and $C(x)$ assigned, say, 1, 0 to wires α, β, then the

Program $F_g^{\Lambda_\alpha, 1\oplus\Lambda_\beta}[\mathsf{mask}](x)$

Constants: $C, g, k_\gamma, \mathsf{td}_\gamma$.

1. Evaluate $C(x)$ and learn bit assignments $\mathsf{bit}_\alpha, \mathsf{bit}_\beta$ of input wires α, β of gate g.
2. Generate $\widehat{k}_\gamma \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_\gamma, x)$.
3. If $g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) = g(\mathsf{bit}_\alpha, 1\oplus \mathsf{bit}_\beta)$ then output $k_\gamma \oplus \mathsf{mask}$;
4. else output $\widehat{k}_\gamma \oplus \mathsf{mask}$.

Program $F_g^{1\oplus\Lambda_\alpha, \Lambda_\beta}[\mathsf{mask}](x)$

Constants: $C, g, k_\gamma, \mathsf{td}_\gamma$.

1. Evaluate $C(x)$ and learn bit assignments $\mathsf{bit}_\alpha, \mathsf{bit}_\beta$ of input wires α, β of gate g.
2. Generate $\widehat{k}_\gamma \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_\gamma, x)$.
3. If $g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) = g(1\oplus\mathsf{bit}_\alpha, \mathsf{bit}_\beta)$ then output $k_\gamma \oplus \mathsf{mask}$;
4. else output $\widehat{k}_\gamma \oplus \mathsf{mask}$.

Program $F_g^{1\oplus\Lambda_\alpha, 1\oplus\Lambda_\beta}[\mathsf{mask}](x)$

Constants: $C, g, k_\gamma, \mathsf{td}_\gamma$.

1. Evaluate $C(x)$ and learn bit assignments $\mathsf{bit}_\alpha, \mathsf{bit}_\beta$ of input wires α, β of gate g.
2. Generate $\widehat{k}_\gamma \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_\gamma, x)$.
3. If $g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) = g(1\oplus\mathsf{bit}_\alpha, 1\oplus\mathsf{bit}_\beta)$ then output $k_\gamma \oplus \mathsf{mask}$;
4. else output $\widehat{k}_\gamma \oplus \mathsf{mask}$.

Program $\mathsf{Const}^{1\oplus\Lambda_\alpha, 1\oplus\Lambda_\beta}[\mathsf{mask}](x)$.

Pad the program to the size of programs F with dummy gates. Output mask.

Output gates: If g is an output gate, we will the same four functions described above with the exception that we will use the actual value instead of the key in the output. For example, for the function $F_g^{\Lambda_\alpha, 1\oplus\Lambda_\beta}[\mathsf{mask}](x)$, we will output $g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) \oplus \mathsf{mask}$ if $g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) = g(\mathsf{bit}_\alpha, 1\oplus\mathsf{bit}_\beta)$ and $1 \oplus g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) \oplus \mathsf{mask}$ otherwise.

Figure 1: Programs F.

garbled gate would decrypt to $k_\gamma, k_\gamma, k_\gamma$ and $\widehat{k}_\gamma$ (in some order) under keys $k_\alpha, \widehat{k}_\alpha, k_\beta, \widehat{k}_\beta$. If bit assignment was instead $1, 1$, then the garbled gate would decrypt to $k_\gamma, \widehat{k}_\gamma, \widehat{k}_\gamma, \widehat{k}_\gamma$ instead.

Finally, the simulator uses $\mathsf{FEE.Inv}$ to simulate random coins of all 8 ciphertexts per gate.

The simulator presents all randomness of encryption, 8 secret shares s per gate, and a pair of keys per gate as internal state of the garbler.

A note on multiple fan-out. In our construction if a wire w is used as an input to multiple gates, we will reuse the keys (i.e. keys $k_w, \widehat{k}_w$, trapdoors td_w) across all of them. This is possible, as our FEE scheme accomodates reusing a single key to simulate multiple ciphertexts (each instantiated with a different function). Furthermore, a key for a wire can be safely used as the left key in one gate and as an right key in another. This will not result in any increase in key sizes since we encrypt shares of the key using the left and right key separately and the sizes of both the keys and the plaintexts remains the same in every gate.[4]

A sketch of the security proof. Starting from the real execution, we will switch the key $\widehat{k}_w$ of each wire from being honestly generated using Gen to a simulated key, starting from $m-1$-th key[5] and traversing the circuit according to its topological sorting all the way to input keys. Namely, suppose that we want to switch a key $\widehat{k}_{w^*} = k_{w^*}^{1 \oplus \mathsf{bit}_{w^*}}$ to a simulated key. We perform the reduction to security of the FEE scheme as follows. First we generate keys and encryption randomness for each wire $w = w^* + 1, \ldots, m-1$ according to the simulation (i.e generate k_w honestly, but equivocate $\widehat{k}_w$ using trapdoor td_w; randomness is equivocated via $\mathsf{FEE.Adapt}$), and generate keys and encryption randomness for wires $w = 1, \ldots, w^* - 1$ according to the real world (i.e. both keys are generated honestly, randomness of encryption is truly random). Also we generate k_{w^*} and its ciphertexts honestly. Recall that the key corresponding to wire w will be used in generating $2 \cdot \mathsf{fanout}$ ciphertexts where fanout is the number gates that have one of its input wires as w^*. The functions that the adversary needs to provide in the security game of FEE for all challenge encryptions (i.e. encryptions which should be generated under the key $\widehat{k}_{w^*}$), namely $f_1, \ldots, f_{2 \cdot \mathsf{fanout}}$, are set appropriately according to Figure 1. We play FEE security game with an input x and functions $f_1, \ldots, f_{2 \cdot \mathsf{fanout}}$; note that the description of each function F_g contains the trapdoor td_γ for an output wire of the gate g, but since we replace keys with simulated keys in reverse topological order, we would have already switched the (other) key corresponding to the output wire γ to simulated, and therefore trapdoors td_γ are well defined. The challenger in the FEE game on input x and the functions, responds with $2 \cdot \mathsf{fanout}$ challenge ciphertexts, $2 \cdot \mathsf{fanout}$ random coins of the encryption and a single key $\widehat{k}_{w^*}$; these values are either real or simulated. The experiment can be reconstructed by placing these ciphertexts in the garbled gates. Now, depending on how the challenge ciphertexts were generated, the resulting experiments yield the corresponding hybrid experiments.

Point-and-permute. In our construction we use a technique from [MNPS04b], which tells the evaluator which row out of 4 rows should be decrypted, in the same way as described in Section 3. Namely, the garbler should additionally choose a bit λ_w per wire, and encrypt not only the key, but "half of a pointer" to the correct row in the next gate. That is, each encryption is $c^{b_1 \oplus \lambda_\alpha, b_2 \oplus \lambda_\beta} = \mathsf{Enc}(k_c^{g(b_1,b_2)} || g(b_1, b_2) \oplus \lambda_\gamma)$, and keys for input bits now also contain pointers $x_w \oplus \lambda_w$. This way the evaluator always knows a pointer $\Lambda_\alpha = \mathsf{bit}_\alpha \oplus \lambda_\alpha$ and $\Lambda_\beta = \mathsf{bit}_\beta \oplus \lambda_\beta$, and therefore knows that it should decrypt ciphertext $c^{\Lambda_\alpha, \Lambda_\beta}$ in the next gate. We refer to λ_α as the hidden mask and Λ_α as the visible mask.

4.2 Full Description of the scheme

Notation. Let λ be security parameter. Let C be a circuit with description size $|C|$ which takes as inputs n-bit strings. We assume that the circuit outputs a single bit; our construction can be adapted for multiple-bit outputs in a straightforward manner. We denote by $m = n + \mathsf{gates}(C)$ the total number of "different" wires in C, i.e. if a gate has a fan-out more than 1, only one wire is counted. We assume that wires from 1 to n are input wires, and wire m is an output wire. Let $|F|$ be the description size of circuits presented in Figure 1.

[4]If we had relied on encrypting the first message with one key followed by the other key, it could possibly result in growing key sizes.

[5]Recall that m-th key doesn't exist, since m-th wire is an output wire.

Assumptions. We assume there is only one output gate. Our construction can be generalized to multi-bit output circuits in a straightforward manner. We also assume without loss of generality that all gates are fan-in two gates: we do this by changing NOT gates to XOR $\mathbb{1}$ gates and therefore replacing the complete system (AND, OR, NOT) with another complete system (AND, OR, XOR $\mathbb{1}$), where $\mathbb{1}$ is a constant gate. In the garbling scheme we can always implement a constant gate $\mathbb{1}$ by setting an additional input wire which the garbler should always set to 1. That is, to garble input x in the circuit with constant gate $\mathbb{1}$, the garbler should instead garble input $\widetilde{x||1}$ in the circuit without constant gates.

Parameters of the FEE scheme. In Section 3 we showed that there exists an FEE scheme for inputs of size n with key size $\kappa = \lambda n + 1$.

Key generation: $\mathsf{Gen}(1^\lambda, C, r_{\mathsf{Gen}})$ takes as input security parameter, the circuit, and randomness r_{Gen} of size $2(m-1)\kappa + m - 1$. It interprets this randomness as $2(m-1)$ random FEE keys $(k_1^0, \ldots, k_{m-1}^0, k_1^1, \ldots, k_{m-1}^1)$ of size κ (a pair of keys per wire, except the output wire of C) and $m-1$ random bits λ_w. Output wires of the same output gate are assigned the same pair of keys and the same values λ_w. Finally the program outputs the garbling key $K = (k_1^0, \ldots, k_{m-1}^0, k_1^1, \ldots, k_{m-1}^1, \lambda_1, \ldots, \lambda_{m-1})$.

Circuit garbling: $\mathsf{Garble}_{\mathsf{Prog}}(K, C; r_{\mathsf{Garble}})$ takes as input the garbling key K (which it interprets as $K = (k_1^0, \ldots, k_{m-1}^0, k_1^1, \ldots, k_{m-1}^1, \lambda_1, \ldots, \lambda_{m-1})$), the circuit C to be garbled, and randomness r_{Garble} which it interprets as random strings rg_g used to encrypt each gate g. Each rg_g consists of randomness $r^{00}_{g,\mathsf{left}}, r^{01}_{g,\mathsf{left}}, r^{10}_{g,\mathsf{left}}, r^{11}_{g,\mathsf{left}}, r^{00}_{g,\mathsf{right}}, r^{01}_{g,\mathsf{right}}, r^{10}_{g,\mathsf{right}}, r^{11}_{g,\mathsf{right}}$, as well as 4 random values $s^{00}_{g,\mathsf{right}}, s^{01}_{g,\mathsf{right}}, s^{10}_{g,\mathsf{right}}, s^{11}_{g,\mathsf{right}}$.

The program sets $\mathsf{params} = (|F|, n, \kappa + 1)$. Next for every gate g with input wires α, β and output wires γ it prepares the following 4 plaintexts:

$$\begin{aligned}
M_g^{00} &= k_\gamma^{g(0,0)} || g(0,0) \oplus \lambda_\gamma, & s^{00}_{g,\mathsf{left}} &= M_g^{00} \oplus s^{00}_{g,\mathsf{right}} \\
M_g^{01} &= k_\gamma^{g(0,1)} || g(0,1) \oplus \lambda_\gamma, & s^{01}_{g,\mathsf{left}} &= M_g^{01} \oplus s^{01}_{g,\mathsf{right}} \\
M_g^{10} &= k_\gamma^{g(1,0)} || g(1,0) \oplus \lambda_\gamma, & s^{10}_{g,\mathsf{left}} &= M_g^{10} \oplus s^{10}_{g,\mathsf{right}} \\
M_g^{11} &= k_\gamma^{g(1,1)} || g(1,1) \oplus \lambda_\gamma, & s^{11}_{g,\mathsf{left}} &= M_g^{11} \oplus s^{11}_{g,\mathsf{right}}.
\end{aligned}$$

Then the program encrypts each plaintext under a pair of keys as follows:

$$\begin{aligned}
c^{\lambda_\alpha, \lambda_\beta}_{g,\mathsf{left}} &= \mathsf{FEE.Enc}_{k^0_\alpha}(\mathsf{params}, s^{00}_{g,\mathsf{left}}; r^{00}_{g,\mathsf{left}}), & c^{\lambda_\alpha, \lambda_\beta}_{g,\mathsf{right}} &= \mathsf{FEE.Enc}_{k^0_\beta}(\mathsf{params}, s^{00}_{g,\mathsf{left}}; r^{00}_{g,\mathsf{right}}), \\
c^{\lambda_\alpha, 1\oplus\lambda_\beta}_{g,\mathsf{left}} &= \mathsf{FEE.Enc}_{k^0_\alpha}(\mathsf{params}, s^{01}_{g,\mathsf{left}}; r^{01}_{g,\mathsf{left}}), & c^{\lambda_\alpha, 1\oplus\lambda_\beta}_{g,\mathsf{right}} &= \mathsf{FEE.Enc}_{k^1_\beta}(\mathsf{params}, s^{01}_{g,\mathsf{left}}; r^{01}_{g,\mathsf{right}}), \\
c^{1\oplus\lambda_\alpha, \lambda_\beta}_{g,\mathsf{left}} &= \mathsf{FEE.Enc}_{k^1_\alpha}(\mathsf{params}, s^{10}_{g,\mathsf{left}}; r^{10}_{g,\mathsf{left}}), & c^{1\oplus\lambda_\alpha, \lambda_\beta}_{g,\mathsf{right}} &= \mathsf{FEE.Enc}_{k^0_\beta}(\mathsf{params}, s^{10}_{g,\mathsf{left}}; r^{10}_{g,\mathsf{right}}), \\
c^{1\oplus\lambda_\alpha, 1\oplus\lambda_\beta}_{g,\mathsf{left}} &= \mathsf{FEE.Enc}_{k^1_\alpha}(\mathsf{params}, s^{11}_{g,\mathsf{left}}; r^{11}_{g,\mathsf{left}}), & c^{1\oplus\lambda_\alpha, 1\oplus\lambda_\beta}_{g,\mathsf{right}} &= \mathsf{FEE.Enc}_{k^1_\beta}(\mathsf{params}, s^{11}_{g,\mathsf{left}}; r^{11}_{g,\mathsf{right}}).
\end{aligned}$$

The program sets each garbled gate G_g to be

$$G_g = \begin{cases} (c^{00}_{g,\mathsf{left}}, & c^{00}_{g,\mathsf{right}}), \\ (c^{01}_{g,\mathsf{left}}, & c^{01}_{g,\mathsf{right}}), \\ (c^{10}_{g,\mathsf{left}}, & c^{10}_{g,\mathsf{right}}), \\ (c^{11}_{g,\mathsf{left}}, & c^{11}_{g,\mathsf{right}}) \end{cases}$$

in this order. It outputs the garbled circuit $\widetilde{C} = \{G_g\}_{g \in [\mathsf{gates}(C)]}$.

Bit-decomposable input garbling: $\mathsf{Garble}_{\mathsf{Inp}}(K, x_i, i)$ takes as input the garbling key K (which it interprets as $K = (k_1^0, \ldots, k_{m-1}^0, k_1^1, \ldots, k_{m-1}^1, \lambda_1, \ldots, \lambda_{m-1})$), and i-th input bit x_i together with its position i. The program outputs a garbled i-th bit $\tilde{x}_i = (k_i^{x_1}, \lambda_i \oplus x_i)$.

Evaluation: $\mathsf{Eval}(\widetilde{C}, \widetilde{x})$ works by evaluating keys $k_w^{\mathsf{bit}(w)}$ (where $\mathsf{bit}(w)$ is a bit assigned to wire w by the computation $C(x)$), going from input gates to an output gate. Assume the evaluator already knows

Program $F_g^{\Lambda_\alpha, 1\oplus\Lambda_\beta}[\mathsf{mask}](x)$

Constants: $C, g, k_\gamma, \mathsf{td}_\gamma, \Lambda_\gamma$.

1. Evaluate $C(x)$ and learn bit assignments $\mathsf{bit}_\alpha, \mathsf{bit}_\beta$ of input wires α, β of gate g.
2. Generate $\widehat{k}_\gamma \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_\gamma, x)$.
3. If $g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) = g(\mathsf{bit}_\alpha, 1 \oplus \mathsf{bit}_\beta)$ then set $M_g^{\Lambda_\alpha, 1\oplus\Lambda_\beta} = k_\gamma || \Lambda_\gamma$;
4. else set $M_g^{\Lambda_\alpha, 1\oplus\Lambda_\beta} = \widehat{k}_\gamma || 1 \oplus \Lambda_\gamma$.
5. Output $M_g^{\Lambda_\alpha, 1\oplus\Lambda_\beta} \oplus \mathsf{mask}$.

Program $F_g^{1\oplus\Lambda_\alpha, \Lambda_\beta}[\mathsf{mask}](x)$

Constants: $C, g, k_\gamma, \mathsf{td}_\gamma, \Lambda_\gamma$.

1. Evaluate $C(x)$ and learn bit assignments $\mathsf{bit}_\alpha, \mathsf{bit}_\beta$ of input wires α, β of gate g.
2. Generate $\widehat{k}_\gamma \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_\gamma, x)$.
3. If $g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) = g(1 \oplus \mathsf{bit}_\alpha, \mathsf{bit}_\beta)$ then set $M_g^{1\oplus\Lambda_\alpha, \Lambda_\beta} = k_\gamma || \Lambda_\gamma$;
4. else set $M_g^{1\oplus\Lambda_\alpha, \Lambda_\beta} = \widehat{k}_\gamma || 1 \oplus \Lambda_\gamma$.
5. Output $M_g^{1\oplus\Lambda_\alpha, \Lambda_\beta} \oplus \mathsf{mask}$.

Program $F_g^{1\oplus\Lambda_\alpha, 1\oplus\Lambda_\beta}[\mathsf{mask}](x)$

Constants: $C, g, k_\gamma, \mathsf{td}_\gamma, \Lambda_\gamma$.

1. Evaluate $C(x)$ and learn bit assignments $\mathsf{bit}_\alpha, \mathsf{bit}_\beta$ of input wires α, β of gate g.
2. Generate $\widehat{k}_\gamma \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_\gamma, x)$.
3. If $g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) = g(1 \oplus \mathsf{bit}_\alpha, 1 \oplus \mathsf{bit}_\beta)$ then set $M_g^{1\oplus\Lambda_\alpha, 1\oplus\Lambda_\beta} = k_\gamma || \Lambda_\gamma$;
4. else set $M_g^{1\oplus\Lambda_\alpha, 1\oplus\Lambda_\beta} = \widehat{k}_\gamma || 1 \oplus \Lambda_\gamma$.
5. Output $M_g^{1\oplus\Lambda_\alpha, 1\oplus\Lambda_\beta} \oplus \mathsf{mask}$.

Program $\mathsf{Const}[\mathsf{const}](x)$.

The program is padded to the size of programs F and is the function that outputs the constant const.

Figure 2: Functions used in FEE simulation

keys and external indices k_α, Λ_α and k_β, Λ_β for input wires of a gate. Then it sets the key and external index k_γ, Λ_γ of the output wire to be $(k_\gamma, \Lambda_\gamma) \leftarrow \mathsf{FEE.Dec}_{k_\alpha}(c_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta}) \oplus \mathsf{FEE.Dec}_{k_\beta}(c_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta})$ and proceeds to the next gate. After decrypting the output gate, the evaluator learns the output bit of the circuit.

Simulation:

Simulating garbled circuit and garbled input. $\mathsf{Sim}_1(1^\lambda, C)$ uses its knowledge of C to determine the size of input n and the number of "different" wires m. It sets $\kappa = \lambda n + 1$ to be the size of the key. Then for each wire w of C except for the output wire the simulator chooses:

- random κ-bit FEE key k_w;

- random bit Λ_w;
- an FEE trapdoor $\mathsf{td}_w \leftarrow \mathsf{FEE.SimTrap}(1^\lambda, n)$.

Next the simulator garbles each gate g as follows: it sets $\mathsf{params} = (|F|, n, \kappa+1)$ Then it computes

$$\begin{array}{llllll}
\text{row } (\Lambda_\alpha, \Lambda_\beta): & c_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta} & = \mathsf{FEE.Enc}_{k_\alpha} & (\mathsf{params}, & s_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta} & ; r_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta}), \\
 & c_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta} & = \mathsf{FEE.Enc}_{k_\beta} & (\mathsf{params}, & s_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta} & ; r_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta}) \\
\text{row } (\Lambda_\alpha, 1\oplus\Lambda_\beta): & c_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} & = \mathsf{FEE.Enc}_{k_\alpha} & (\mathsf{params}, & s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} & ; r_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}), \\
 & c_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} & = \mathsf{FEE.SimEnc} & (F_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}] & & ; r_{\mathsf{Sim},g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}) \\
\text{row } (1\oplus\Lambda_\alpha, \Lambda_\beta): & c_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta} & = \mathsf{FEE.SimEnc} & (F_g^{1\oplus\Lambda_\alpha,\Lambda_\beta}[s_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}] & & ; r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}), \\
 & c_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta} & = \mathsf{FEE.Enc}_{k_\beta} & (\mathsf{params}, & s_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta} & ; r_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}) \\
\text{row } (1\oplus\Lambda_\alpha, 1\oplus\Lambda_\beta): & c_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} & = \mathsf{FEE.SimEnc} & (\mathsf{Const}[s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}] & & ; r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}), \\
 & c_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} & = \mathsf{FEE.SimEnc} & (F_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}] & & ; r_{\mathsf{Sim},g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}).
\end{array}$$

where $s_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta} = (k_\gamma||\Lambda_\gamma) \oplus s_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta}$, and $s_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta}$ are chosen at random.

The simulator sets each garbled gate G_g to be

$$G_g = (c_{g,\mathsf{left}}^{00}, c_{g,\mathsf{right}}^{00}, c_{g,\mathsf{left}}^{01}, c_{g,\mathsf{right}}^{01}, c_{g,\mathsf{left}}^{10}, c_{g,\mathsf{right}}^{10}, c_{g,\mathsf{left}}^{11}, c_{g,\mathsf{right}}^{11}),$$

in this order. It outputs the garbled circuit $\widetilde{C} = \{G_g\}_{g\in[\mathsf{gates}(C)]}$. It sets simulated garbled input $\widetilde{x} = ((k_1, \Lambda_1), \ldots, (k_n, \Lambda_n))$.

The simulator sets its state to be $k_1, \ldots, k_{m-1}, \Lambda_1, \ldots, \Lambda_{m-1}, \mathsf{td}_1, \ldots, \mathsf{td}_{m-1}$, as well as G_g, $r_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta}$, $r_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta}$, $r_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}$, $r_{\mathsf{Sim},g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}$, $r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}$, $r_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}$, $r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}$, $r_{\mathsf{Sim},g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}$, $s_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta}$, $s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}$, $s_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}$, $s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}$, for every gate g of C.

Simulating internal state of the garbler. $\mathsf{Sim}_2(x, \mathsf{state})$ first sets $\widehat{k}_w \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_w, x)$ for wires $w = 1, \ldots, m-1$. Next for every gate g it sets randomness of encryption to be

$$\begin{aligned}
r_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} &\leftarrow \mathsf{FEE.Adapt}(F_g^{\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}], r_{\mathsf{Sim},g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}, x), \\
r_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta} &\leftarrow \mathsf{FEE.Adapt}(F_g^{1\oplus\Lambda_\alpha,\Lambda_\beta}[s_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}], r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}, x), \\
r_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} &\leftarrow \mathsf{FEE.Adapt}(\mathsf{Const}[s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}], r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}, x), \\
r_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} &\leftarrow \mathsf{FEE.Adapt}(F_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}], r_{\mathsf{Sim},g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}, x),
\end{aligned}$$

The simulator sets $k_w^{\mathsf{bit}_w} \leftarrow k_w, k_w^{1\oplus\mathsf{bit}_w} \leftarrow \widehat{k}_w$ for wires $w = 1, \ldots, m-1$ (where bit_w is the bit assigned to wire w by the computation $C(x)$). In addition, for every input wire $w = 1, \ldots, m-1$ it sets $\lambda_w = \Lambda_w \oplus \mathsf{bit}_w$.

Next it sets

$$\begin{aligned}
s_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} &= F_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}](x), \\
s_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} &= F_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}](x).
\end{aligned}$$

The simulator outputs $(k_w^0, k_w^1, \lambda_w)$ for each wire $w = 1, \ldots, m-1$, and $r_{g,\mathsf{left}}^{00}, r_{g,\mathsf{left}}^{01}, r_{g,\mathsf{left}}^{10}, r_{g,\mathsf{left}}^{11}$, $r_{g,\mathsf{right}}^{00}, r_{g,\mathsf{right}}^{01}, r_{g,\mathsf{right}}^{10}, r_{g,\mathsf{right}}^{11}$, together with $s_{g,\mathsf{right}}^{00}, s_{g,\mathsf{right}}^{01}, s_{g,\mathsf{right}}^{10}, s_{g,\mathsf{right}}^{11}$, for each gate g as internal state of the garbler.

4.3 Proof

Correctness. Correctness follows from correctness of underlying encryption similar to the correctness of Yao garbled circuit. Namely, it can be shown by induction that at each step the evaluator gets the correct key $k_\gamma^{\mathsf{bit}_\gamma}$ and the correct pointer Λ_γ for the next gate' row.

Security. We consider a sequence of hybrid experiments starting from real world experiment where $\widetilde{C}, \widetilde{x}, r$ are generated honestly to the simulated experiment where $\widetilde{C}, \widetilde{x}$ are simulated given only C and output $y = C(x)$, and r is equivocated to x. We consider $m-1$ intermediate hybrids, where in hybrid H_i we switch the key $k_{m-i}^{\mathsf{bit}_{m-i}}$ from real to simulated. Here we assume that wires are sorted according to the topological order of the circuit, i.e. that output wires of each gate have larger index than both input wires of that gate (note that our notation $1, \ldots, n$ for input wires and m for an output wire is consistent with topological order).

Hybrid H_0. In this hybrid we change how the permutation of ciphertexts is generated, without changing the distribution of the hybrid. Roughly speaking, instead of choosing λ_w at random and setting $\Lambda_w = \lambda_w \oplus \mathsf{bit}_w$, we choose Λ_w at random and set $\lambda_w = \Lambda_w \oplus \mathsf{bit}_w$, which clearly results in the same distributions. More precisely, recall that in the real execution we choose random λ_w for each wire w and set each ciphertext $c^{b_1 \oplus \lambda_\alpha, b_2 \oplus \lambda_\beta}$ to be an encryption of $k_\gamma^{g(b_1,b_2)} || \lambda_\gamma \oplus g(b_1, b_2)$, for all bits b_1, b_2. In this hybrid we instead choose a random bit Λ_w for each wire and set each ciphertext $c^{b_1 \oplus \Lambda_\alpha, b_2 \oplus \Lambda_\beta}$ to be an encryption of $k_\gamma^{g(\mathsf{bit}_\alpha \oplus b_1, \mathsf{bit}_\beta \oplus b_2)} || \Lambda_\gamma \oplus g(\mathsf{bit}_\alpha \oplus b_1, \mathsf{bit}_\beta \oplus b_2)$. When internal state of the garbler needs to be presented, we set each λ_w to be $\Lambda_w \oplus \mathsf{bit}_w$.

Hybrid 0 is identical to the real experiment.

Hybrid H_i, $i = 1, \ldots, m-1$. Denote the wire number $m - i$ by w^*. We refer to $k_{w^*}^{1 \oplus \mathsf{bit}_{w^*}}$ as the *challenge key*, and to all (single-encryption) ciphertexts encrypted under this key as *challenge* ciphertexts. The randomness and plaintexts of challenge ciphertexts are denoted as *challenge randomness*, and *challenge plaintexts*. In this hybrid we switch $k_{w^*}^{1 \oplus \mathsf{bit}_{w^*}}$ from real to simulated.

We choose $m-1$ random bits Λ_w. For all wires $w = 1, \ldots, w^*$ the we choose FEE keys k_w^0, k_w^1 at random. For all wires $w = w^* + 1, \ldots, m-1$ we choose the key $k_w^{\mathsf{bit}_w}$, which we denote as k_w, at random and choose the trapdoor for the other key as $\mathsf{td} \leftarrow \mathsf{FEE.SimTrap}(r_{w,\mathsf{SimTrap}})$ with random coins $r_{w,\mathsf{SimTrap}}$. For each $w = w^* + 1, \ldots, m-1$ we set the other key $k_\alpha^{1 \oplus \mathsf{bit}_\alpha} \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_w, x)$; we denote it as $\widehat{k}_w$. Next we generate the garbled circuit as follows: for each gate g we set

$$\begin{aligned}
M_g^{\Lambda_\alpha, \Lambda_\beta} &= k_\gamma^{g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta)} || \Lambda_\gamma, \\
M_g^{\Lambda_\alpha, 1 \oplus \Lambda_\beta} &= k_\gamma^{g(\mathsf{bit}_\alpha, 1 \oplus \mathsf{bit}_\beta)} || \Lambda_\gamma \oplus g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) \oplus g(\mathsf{bit}_\alpha, 1 \oplus \mathsf{bit}_\beta), \\
M_g^{1 \oplus \Lambda_\alpha, \Lambda_\beta} &= k_\gamma^{g(1 \oplus \mathsf{bit}_\alpha, \mathsf{bit}_\beta)} || \Lambda_\gamma \oplus g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) \oplus g(1 \oplus \mathsf{bit}_\alpha, \mathsf{bit}_\beta), \\
M_g^{1 \oplus \Lambda_\alpha, 1 \oplus \Lambda_\beta} &= k_\gamma^{g(1 \oplus \mathsf{bit}_\alpha, 1 \oplus \mathsf{bit}_\beta)} || \Lambda_\gamma \oplus g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) \oplus g(1 \oplus \mathsf{bit}_\alpha, 1 \oplus \mathsf{bit}_\beta).
\end{aligned}$$

We choose random values $s_g^{0,0}, s_g^{0,1}, s_g^{1,0}, s_g^{1,1}$ and set

$$S_g^{0,0} = s_g^{0,0} \oplus M^{0,0},$$

$$S_g^{0,1} = s_g^{0,1} \oplus M^{0,1},$$

$$S_g^{1,0} = s_g^{1,0} \oplus M^{1,0},$$

$$S_g^{1,1} = s_g^{1,1} \oplus M^{1,1}.$$

Next we send an input and functions to the challenger of the FEE security game. For this, we set the input to be x and functions $f_1, \ldots, f_l$ to be functions corresponding to ciphertexts encrypted under key

$\widehat{k}_{w^*}$ (since each gate with input wire w^* contains two such ciphertexts, there are $l = 2 \cdot \mathsf{fanout}$ functions in total, where fanout is fanout of the gate with output wires w^*). More specifically, we prepare the following functions:

- For each gate g where $\widehat{k}_{w^*}$ is used to encrypt left ciphertexts, prepare functions $F_g^{1\oplus\Lambda_\alpha,\Lambda_\beta}[s_g^{1\oplus\Lambda_\alpha,\Lambda_\beta}]$ and $\mathsf{Const}[S_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}]$.
- For each gate g where $\widehat{k}_{w^*}$ is used to encrypt right ciphertexts, prepare functions $F_g^{\Lambda_\alpha,1\oplus\Lambda_\beta}[S_g^{\Lambda_\alpha,1\oplus\Lambda_\beta}]$, $F_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}[S_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}]$.

Note that each function F_g has $C, g, k_\gamma, \mathsf{td}_\gamma, \Lambda_\gamma$ (where γ is an output wire of g) in its description. Since we switch keys to simulated according to the topological order of the circuit, output keys $\widehat{k}_\gamma$ of gates where w^* is an input wire are already switched to simulated, which means that their trapdoor td_γ, and therefore the desecription if each required function F, is well defined.

The challenger of FEE security game responds with a key $\widehat{k}_{w^*}$, l ciphertexts and l randomness of encryption, which are either real (encrypting $F(x)$) or simulated. More specifically, we get the following values from the challenger:

- For each gate g where $\widehat{k}_{w^*}$ is used to encrypt left ciphertexts:
  - In the real case, we get
$$\begin{aligned} c_{g,\mathsf{left}}^{1\oplus\Lambda_{w^*},\Lambda_\beta} &= \mathsf{FEE.Enc}_{\widehat{k}_{w^*}}(\mathsf{params}, S^{1\oplus\Lambda_{w^*},\Lambda_\beta}; r_{g,\mathsf{left}}^{1\oplus\Lambda_{w^*},\Lambda_\beta}), \\ c_{g,\mathsf{left}}^{1\oplus\Lambda_{w^*},1\oplus\Lambda_\beta} &= \mathsf{FEE.Enc}_{\widehat{k}_{w^*}}(\mathsf{params}, S_g^{1\oplus\Lambda_{w^*},1\oplus\Lambda_\beta}; r_{g,\mathsf{left}}^{1\oplus\Lambda_{w^*},1\oplus\Lambda_\beta}), \end{aligned}$$
    We also get both randomness of encryption $r_{g,\mathsf{left}}^{1\oplus\Lambda_{w^*},\Lambda_\beta}$ and $r_{g,\mathsf{left}}^{1\oplus\Lambda_{w^*},1\oplus\Lambda_\beta}$.
  - In the simulated case, we get
$$\begin{aligned} c_{g,\mathsf{left}}^{1\oplus\Lambda_{w^*},\Lambda_\beta} &= \mathsf{FEE.SimEnc}(F_g^{1\oplus\Lambda_{w^*},\Lambda_\beta}[s_g^{1\oplus\Lambda_{w^*},\Lambda_\beta}], \mathsf{td}_{w^*}; r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_{w^*},\Lambda_\beta}) \\ c_{g,\mathsf{left}}^{1\oplus\Lambda_{w^*},1\oplus\Lambda_\beta} &= \mathsf{FEE.SimEnc}(\mathsf{Const}[S_g^{1\oplus\Lambda_{w^*},1\oplus\Lambda_\beta}], \mathsf{td}_{w^*}; r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_{w^*},1\oplus\Lambda_\beta}). \end{aligned}$$
    We also get both randomness of encryption
$$\begin{aligned} r_{g,\mathsf{left}}^{1\oplus\Lambda_{w^*},\Lambda_\beta} &\leftarrow \mathsf{FEE.Adapt}(F_g^{1\oplus\Lambda_{w^*},\Lambda_\beta}[s_g^{1\oplus\Lambda_{w^*},\Lambda_\beta}], r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_{w^*},\Lambda_\beta}), x), \\ r_{g,\mathsf{left}}^{1\oplus\Lambda_{w^*},1\oplus\Lambda_\beta} &\leftarrow \mathsf{FEE.Adapt}(\mathsf{Const}[S_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}], r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_{w^*},1\oplus\Lambda_\beta}, x). \end{aligned}$$
- For each gate g where $\widehat{k}_{w^*}$ is used to encrypt right ciphertexts:
  - In the real case, we get
$$\begin{aligned} c_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_{w^*}} &= \mathsf{FEE.Enc}_{\widehat{k}_{w^*}}(\mathsf{params}, s^{\Lambda_\alpha,1\oplus\Lambda_{w^*}}; r_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_{w^*}}), \\ c_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_{w^*}} &= \mathsf{FEE.Enc}_{\widehat{k}_{w^*}}(\mathsf{params}, s_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_{w^*}}; r_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_{w^*}}), \end{aligned}$$
    We also get both randomness of encryption $r_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_{w^*}}$ and $r_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_{w^*}}$.

- In the simulated case, we get

$$
\begin{aligned}
c_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_{w^*}} &= \mathsf{FEE.SimEnc}(F_g^{\Lambda_\alpha,1\oplus\Lambda_{w^*}}[S_g^{\Lambda_\alpha,1\oplus\Lambda_{w^*}}], \mathsf{td}_{w^*}; r_{\mathsf{Sim},g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_{w^*}}),\\
c_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_{w^*}} &= \mathsf{FEE.SimEnc}(F_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_{w^*}}[S_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_{w^*}}], \mathsf{td}_{w^*}; r_{\mathsf{Sim},g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_{w^*}}).
\end{aligned}
$$

  We also get both randomness of encryption

$$
\begin{aligned}
r_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_{w^*}} &\leftarrow \mathsf{FEE.Adapt}(F_g^{\Lambda_\alpha,1\oplus\Lambda_{w^*}}[S_g^{\Lambda_\alpha,1\oplus\Lambda_{w^*}}], r_{\mathsf{Sim},g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_{w^*}}), x),\\
r_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_{w^*}} &\leftarrow \mathsf{FEE.Adapt}(F_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_{w^*}}[S_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_{w^*}}], r_{\mathsf{Sim},g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_{w^*}}, x).
\end{aligned}
$$

- Finally, we get the key $\widehat{k}_{w^*}$, which is either randomly chosen or simulated as $\widehat{k}_{w^*} \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_{w^*}, x)$.

We then generate the garbled gate as follows:

1. The first pair of left encryptions (under active key k_α) is generated as follows:

$$
\begin{aligned}
c_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta} &= \mathsf{FEE.Enc}_{k_\alpha}(\mathsf{params}, S^{\Lambda_\alpha,\Lambda_\beta}; r_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta}),\\
c_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} &= \mathsf{FEE.Enc}_{k_\alpha}(\mathsf{params}, S_g^{\Lambda_\alpha,1\oplus\Lambda_\beta}; r_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}).
\end{aligned}
$$

2. Then we generate the second pair of left encryptions (under inactive key $\widehat{k}_\alpha$) as follows:

   - If $\alpha > w^*$ (i.e. the key $\widehat{k}_\alpha$ is already simulated), then we set:

$$
\begin{aligned}
c_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta} &= \mathsf{FEE.SimEnc}(F_g^{1\oplus\Lambda_\alpha,\Lambda_\beta}[s_g^{1\oplus\Lambda_\alpha,\Lambda_\beta}], \mathsf{td}_\alpha; r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}),\\
c_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} &= \mathsf{FEE.SimEnc}(\mathsf{Const}[S_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}], \mathsf{td}_\alpha; r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}),
\end{aligned}
$$

     and set equivocated randomness of encryption to be

$$
\begin{aligned}
r_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta} &= \mathsf{FEE.Adapt}(F_g^{1\oplus\Lambda_\alpha,\Lambda_\beta}[s_g^{1\oplus\Lambda_\alpha,\Lambda_\beta}], r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}, x),\\
r_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} &= \mathsf{FEE.Adapt}(\mathsf{Const}_g[S_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}], r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}, x),
\end{aligned}
$$

   - If $\alpha < w^*$ (i.e. the key $\widehat{k}_\alpha$ is still honestly generated), then we instead generate these two ciphertexts as

$$
\begin{aligned}
c_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta} &= \mathsf{FEE.Enc}_{\widehat{k}_\alpha}(\mathsf{params}, S^{1\oplus\Lambda_\alpha,\Lambda_\beta}; r_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}),\\
c_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} &= \mathsf{FEE.Enc}_{\widehat{k}_\alpha}(\mathsf{params}, S_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}; r_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}).
\end{aligned}
$$

   - If $\alpha = w^*$, then we use challenge ciphertexts $c_{g,\mathsf{left}}^{1\oplus\Lambda_{w^*},\Lambda_\beta}$, $c_{g,\mathsf{left}}^{1\oplus\Lambda_{w^*},1\oplus\Lambda_\beta}$.

3. Similarly, we generate a pair of right ciphertexts under active key k_β as follows:

$$
\begin{aligned}
c_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta} &= \mathsf{FEE.Enc}_{k_\beta}(\mathsf{params}, s^{\Lambda_\alpha,\Lambda_\beta}; r_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta}),\\
c_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} &= \mathsf{FEE.Enc}_{k_\beta}(\mathsf{params}, s_g^{\Lambda_\alpha,1\oplus\Lambda_\beta}; r_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}).
\end{aligned}
$$

4. Then we generate the second pair of right encryptions (under inactive key $\widehat{k}_\beta$) as follows:

   - If $\beta > w^*$ (i.e. the key $\widehat{k}_\beta$ is already simulated), then we set:

$$c_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} = \mathsf{FEE.SimEnc}(F_g^{\Lambda_\alpha,1\oplus\Lambda_\beta}[S_g^{\Lambda_\alpha,1\oplus\Lambda_\beta}], \mathsf{td}_\beta; r_{\mathsf{Sim},g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}),$$
$$c_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} = \mathsf{FEE.SimEnc}(F_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}[S_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}], \mathsf{td}_\beta; r_{\mathsf{Sim},g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}),$$

     and set equivocated randomness of encryption to be

$$r_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} = \mathsf{FEE.Adapt}(F_g^{\Lambda_\alpha,1\oplus\Lambda_\beta}[S_g^{\Lambda_\alpha,1\oplus\Lambda_\beta}], r_{\mathsf{Sim},g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}, x),$$
$$r_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} = \mathsf{FEE.Adapt}(F_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}[S_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}], r_{\mathsf{Sim},g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}, x).$$

   - If $\beta < w^*$ (i.e. the key $\widehat{k}_\beta$ is still honestly generated), then we instead generate these two ciphertexts as

$$c_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} = \mathsf{FEE.Enc}_{\widehat{k}_\beta}(\mathsf{params}, s_g^{\Lambda_\alpha,1\oplus\Lambda_\beta}; r_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}),$$
$$c_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} = \mathsf{FEE.Enc}_{\widehat{k}_\beta}(\mathsf{params}, s_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}; r_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}).$$

   - If $\beta = w^*$, then we use challenge ciphertexts $c_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_{w^*}}, c_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_{w^*}}$.

We set each garbled gate G_g to be 8 ciphertexts $c_{g,\mathsf{left}}^{b_1,b_2}, c_{g,\mathsf{right}}^{b_1,b_2}$ generated above (for all bits b_1, b_2), and we set the garbled circuit $\widetilde{C}$ to be $\{G_g\}_{g\in[\mathsf{gates}(C)]}$. We set the garbled input to be $k_1, \ldots, k_n$. We set the internal state of the garbler to be:

- Keys $k_1, \ldots, k_{m-1}, \widehat{k}_1, \ldots, \widehat{k}_{m-1}$;
- Encryption randomness $r_{g,\mathsf{left}}^{b_1,b_2}, r_{g,\mathsf{right}}^{b_1,b_2}$, for each gate g and each pair of bits b_1, b_2.
- Masks $s_g^{b_1,b_2}$ for every gate g and each pair of bits b_1, b_2.
- Random bits $\lambda_w = \Lambda_w \oplus \mathsf{bit}_w$.

It follows directly from our construction that if the challenge ciphertexts were honestly generated according to FEE.Gen and FEE.Enc, the resulting experiment would be H_{w^*-1} and if they are simulated the experiment would be H_{w^*}. Therefore indistinguishability of H_{w^*-1} and H_{w^*} directly reduces to the security game of the underlying FEE scheme.

Hybrid H_m. In this hybrid we change how masks are generated, without changing the distribution of the hybrid $m-1$. Since all keys $\widehat{k}_w$ are now simulated, each double encryption depends on only one value s (S, respectively), but not on all three values $S = s \oplus M$. Thus, each encryption can be generated by the simulator who doesn't know x (and therefore M), but first picks random s (S, respectively) and later opens $s = S \oplus M$. Therefore the generation of a garbled circuit is now independent of x (and only depends on $y = C(x)$, and therefore can be simulated before x is known.

5 Application 1: Two-Message Adaptive 2PC

In this section, we provide our main result for adaptive honest-but-curious (semi-honest) corruptions in the two-party setting. We show that the Yao two-message two-party protocol for static honest-but-curious corruptions [Yao86] gives adaptively secure two party computation in the plain model, if we replace the underlying primitives, i.e. garbling and oblivious transfer, with Equivocal Garbling and adaptively secure honest-but-curious oblivious-transfer, and encrypt the communication under non-committing encryption.

Assume one party (the garbler G) has input x and the other party (the evaluator E) has input y; both inputs have length n. We describe the protocol which allows E to learn output $z = f(x, y)$, where f is a deterministic function. To allow G also learn the output, parties can run in parallel another instance of the protocol with reversed roles.

Adaptively secure version of Yao'86 protocol. Let $(\mathsf{Garble}_{\mathsf{Prog}}, \mathsf{Garble}_{\mathsf{Inp}}, \mathsf{Eval})$ be a bit-decomposable equivocal garbling scheme, (R, S, E) be 2-message adaptively secure oblivious transfer, and $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a non-committing encryption (NCE). The protocol proceeds as follows:

1. E sends G $\mathsf{OT}_{1,1} = R(y_1; s_1), \ldots, \mathsf{OT}_{1,n} = R(y_n, s_n)$ for random coins $s_1, \ldots, s_n$, and public key pk of NCE, sampled as $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(r_{\mathsf{Gen}})$ for random r_{Gen};
2. G chooses a random garbling key K and garbles the circuit $\tilde{f} \leftarrow \mathsf{Garble}_{\mathsf{Prog}}(K, f; r)$ using random coins r. Next it garbles its input bit by bit by running $\tilde{x}_i \leftarrow \mathsf{Garble}_{\mathsf{Inp}}(K, x_i, i)$, $i \in [n]$, and garbles each possible bit of E's input by running $\tilde{y}_i^0 \leftarrow \mathsf{Garble}_{\mathsf{Inp}}(K, 0, n+i), \tilde{y}_i^1 \leftarrow \mathsf{Garble}_{\mathsf{Inp}}(K, 1, n+i)$. G sets the plaintext M of NCE to be:
   - the garbled circuit $\tilde{f}$;
   - G's garbled input $\tilde{x}_1, \ldots, \tilde{x}_n$;
   - $\mathsf{OT}_{2,1} = S(\mathsf{OT}_{1,1}, \tilde{y}_1^0, \tilde{y}_1^1; r_1)$, $\ldots$, $\mathsf{OT}_{2,n} = S(\mathsf{OT}_{1,n}, \tilde{y}_n^0, \tilde{y}_n^1; r_n)$ (generated using random coins $r_1, \ldots, r_n$).

   G then encrypts M by setting $c = \mathsf{NCE.Enc}_{\mathsf{pk}}(M; r_{\mathsf{Enc}})$ and sends c to E;

Then E decrypts c using sk of NCE, recovers its garbled input by running $\tilde{y}_i^{y_i} \leftarrow E(\mathsf{OT}_{2,i}, s_i)$, and evaluates $z = \mathsf{Eval}(\tilde{C}, \tilde{x}_1, \ldots, \tilde{x}_n, \tilde{y}_1^{y_1}, \ldots, \tilde{y}_n^{y_n})$.

Theorem 3. *Assume that* $(\mathsf{Garble}_{\mathsf{Prog}}, \mathsf{Garble}_{\mathsf{Inp}}, \mathsf{Eval})$ *is a bit-decomposable equivocal garbling scheme,* (R, S, E) *is an adaptively secure semi-honest oblivious transfer, and* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is a non-committing encryption; all primitives are in the plain model. Then the protocol presented above is a two-party two-message protocol in the plain model, adaptively secure against semi-honest adversaries.*

Proof. Correctness of this protocol follows from correctness of the garbling scheme , oblivious transfer, and NCE. To show security, we show that the simulator can simulate the execution for an arbitrary order of corruptions of possibly both parties. We describe the simulator for each case separately, keeping in mind that the simulator's actions should be independent of the future corruptions.

We assume that the simulator knows the output z in the beginning of the protocol: this can be achieved by instructing E to pick random mask, set it as part of E's input, and run the protocol for the function $z = f'(x, (y, \mathsf{mask})) = f(x, y) \oplus \mathsf{mask}$. This still allows E to recover $f(x, y)$ as $f(x, y) = z \oplus \mathsf{mask}$, but now the output z is uniformly distributed and can be chosen by the simulator in advance; once E is corrupted, the simulator learns $f(x, y)$ and can set the mask accordingly as $\mathsf{mask} = z \oplus f(x, y)$.

Without loss of generality, we can assume that both parties eventually become corrupted (if they don't, the adversary sees strictly less information than in case when they both become corrupted, and therefore security also holds).

- **Case 1: E is corrupted before round 1, G is corrupted any time before round 2.** In this case the simulator learns both inputs x, y and can generate the execution honestly.

- **Case 2: E is corrupted any time after round 1, G is corrupted any time before round 2.** In this case the simulator simulates the first message using OT simulator and NCE simulator, but generates the second message honestly.

  Upon corruption of E, the simulator uses OT simulator to show E's internal state consistent with its input y, and NCE simulator to show that simulated pk was generated honestly.

  Security. We start from a real execution of the protocol.

  In hybrid 1 we change NCE keys and generation randomness $(\mathsf{pk}, \mathsf{sk}, r_{\mathsf{Gen}})$ from real to simulated (the NCE ciphertext is still encrypted honestly, i.e. by executing $\mathsf{NCE.Enc}_{\mathsf{pk}}(M; r_{\mathsf{Enc}})$ with simulated pk); indistinguishability follows from indistinguishability of real and simulated keys of NCE (which is implied by security of NCE).

  In hybrid 2 we change first OT messages to from real to simulated, relying on adaptive security of OT for the case when the receiver is corrupted after it sends the message and the sender is corrupted before the sender sends the message.

- **Case 3: E is corrupted before round 1, G is corrupted after round 2.** In this case the simulator generates the first message honestly, but simulates NCE plaintext M: for this, it runs the simulator for the garbled circuit (where an output of the circuit is set to be an output of the computation, which we assumed to be known to the simulator in advance) and gets simulated $\tilde{f}, \tilde{x}, \tilde{y}$; it simulates second OT messages, setting them to contain a single value $\tilde{y}_i$. It then encrypts this M honestly under NCE.

  Upon corruption of G, the simulator learns x and runs the simulator of garbled circuits to produce a simulated key K and random coins r consistent with input x, y. It uses K to garbled all possible bits $\tilde{y}_i^0, \tilde{y}_i^1$. It also runs the OT simulator to show randomness of G consistent with simulated second OT messages and G's input to the OT $(\tilde{y}_i^0, \tilde{y}_i^1)$.

  Security. We start from a real execution of the protocol.

  In hybrid 1 we change second OT messages $\mathsf{OT}_{2,i}$ (and G's randomness r_i used to generate them) from real to simulated, where each OT message contains a single value $\tilde{y}_i^{y_i}$, i.e. the garbled bit of E's input. Indistinguishability follows from security of OT for the case when the receiver is corrupted before the receiver sends its message, and the sender is corrupted after the sender sends its message.

  In hybrid 2 we change how the garbled circuit is generated: we simulate the garbled circuit $\tilde{f}$ (setting it with the output of the computation, known to the simulator) and the garbled input $\tilde{x}, \tilde{y}$. We also set K, r to be simulated key and randomness for input (x, y). Indistinguishability follows from security of equivocal garbled circuits. We note that it is enough to use security with respect to *selectively* chosen inputs[6], since by the moment the garbled circuit needs to be generated (i.e. round 2), both inputs x, y of the computation are already determined by the environment.

- **Case 4: E is corrupted any time after round 1, G is corrupted after round 2.** In this case the simulator generates both messages as simulated. That is, it generates the first message by simulating first OT messages, and by setting pk to be simulated NCE key. It generates the second message by simulating NCE ciphertext c.

[6]I.e. the adversary has to choose inputs x, y before it sees garbled $\tilde{f}$.

– Case 1: G is corrupted first. Upon corruption of G, the simulator picks the garbling key K at random and garbles $\tilde{f}, \tilde{x}$, and each $\tilde{y}_i^0, \tilde{y}_i^1$ honestly. It generates second OT messages honestly (i.e. containing $\tilde{y}_i^0, \tilde{y}_i^1$). It sets M to be $\tilde{x}, \tilde{f}$, and all second OT messages. It equivocates r_{Enc} such that simulated c looks like a real encryption of M.

  Upon corruption of E, the simulator equivocates sk to M and uses OT simulator to present E's internal state consistent with simulated first OT messages and its input y.

  Security. We start from a real execution of the protocol.

  In hybrid 1 we change how the NCE values are generated: namely, pk, c become simulated, and $r_{\mathsf{Gen}}, \mathsf{sk}, r_{\mathsf{Enc}}$ are equivocated to honestly generated plaintext M. Indistinguishability follows from adaptive security of NCE.

  In hybrid 2 we change first OT messages $\mathsf{OT}_{1,i}$ and E's randomness s_i to simulated. Indistinguishability follows from adaptive security of OT for the case when the receiver is corrupted after it sends its message and the sender is corrupted before the sender sends its message.

– Case 2: E is corrupted first. Upon corruption of E, the simulator first runs the simulator for garbled circuits and gets simulated $\tilde{x}, \tilde{y}, \tilde{f}$. It simulates the second OT messages, such that they contain a single value $\tilde{y}_i$. It sets $M = \tilde{f}, \tilde{x}$, and second OT messages, and equivocates sk to M. It also uses the simulator for adaptive OT to generate E's randomness consistent with its input y and first OT messages.

  Upon corruption of G, the simulator uses the simulator for garbled circuits to simulate the garbling key K and randomness r, consistent with $\tilde{f}, \tilde{x}, \tilde{y}, f, x, y$. It uses K to compute G's input to adaptive OT, i.e. $\tilde{y}_i^0, \tilde{y}_i^1$, and uses OT simulator to produce random coins consistent with this input and simulated second OT messages. Next it equivocates randomness r_{Enc} to M.

  Security. We start from a real execution of the protocol.

  In hybrid 1 we change how OT messages are generated: we simulate both first and second OT messages, setting each OT output to $\tilde{y}_i^{y_i}$ (i.e. garbled bit of E's input). We use adaptive simulator for OT to simulate both parties randomness of OT, such that this randomness is consistent with simulated OT messages and each party's input to OT: namely, with E's input y and G's input $(\tilde{y}_i^0, \tilde{y}_i^1)$.

  In hybrid 2 we change how garbled values are generated: we simulate the garbled circuit $\tilde{f}$ (setting it with the output of the computation, known to the simulator) and the garbled input $\tilde{x}, \tilde{y}$. We also set K, r to be simulated key and randomness for input (x, y). Indistinguishability follows from security of equivocal garbled circuits. Again, we note that it is enough to use security with respect to *selectively* chosen inputs, since by the moment the garbled circuit needs to be generated (i.e. round 2), both inputs x, y of the computation are already determined by the environment.

  Finally, we change NCE values from real to simulated, using adaptive security of NCE.

$\square$

A note on *not* requiring the Garbling scheme to be secure against adaptive choice of inputs. As mentioned in the introduction, an adaptive garbling scheme [BHR12] is a garbling scheme that is secure against an adversary that adaptively chooses its input after seeing the garbled circuit. This notion is not directly related to our notion of equivocal garbling. However, it might seem that our construction requires this stronger property. Consider the corruption scenario where E is corrupted at the end and the G is corrupted before, say at the beginning. Here, the simulator first needs to construct a garbled circuit that is sent from G to E and then later after it receives E's input need to generate keys according to this input and here it might seem that we require our scheme to be secure against an adaptive choice of inputs. This will not be an issue for our construction described above as we are proving security of a two-party protocol for adaptive honest-but-curious corruptions and the inputs for the parties are indeed chosen at the beginning of the computation.[7] This means that when we argue the security via a standard hybrid argument from the real experiment to the simulated experiment, we can assume that in the intermediate hybrids, the simulator has access to the real inputs x, y and thus we will not require the adaptive choice of inputs property to hold for our garbling scheme.

6 Application 2: Constant-round Adaptive Multiparty Computation

In this section, we describe how to adapt our garbling scheme to the multiparty setting.

Theorem 4. *Let f be a deterministic function with n inputs. Assuming the existence of simulatable public-key encryption, there exists a $O(1)$-round multiparty protocol to securely realize f against adaptive honest-but-curious corruption of all parties.*

Proof. On a high-level, our idea is to port the Equivocal Garbling to the multiparty setting in a way similar that standard Garbling is adopted in the approach of Beaver, Micali and Rogaway [BMR90].

Protocol. Let f be a deterministic function that takes inputs $x_1, \dots, x_n$ and outputs $f(x_1, \dots, x_n)$. Let C be the circuit that realizes the function f. Let s be the number of gates and W be the total number of wires. As in our previous descriptions, the gates are numbered so that they are in topological order and the wires are numbered so that the first $|x_1| + \cdots + |x_n|$ wires are assigned to their respective inputs and the last wire, i.e. W is the wire for the output bit. Furthermore, we denote by the indexes $i_1, \dots, i_m$ the wires that carry the input of party P_i. The protocol Π_{BMR} involves parties $P_1, \dots, P_n$ and proceeds in two phases: the first phase is called preprocessing phase and can be executed independent of the parties inputs; the second phase is called online phase.

Preprocessing Phase: The parties in this phase will execute an generic MPC protocol (say, the GMW protocol) for an ideal multiparty functionality that will result in an equivocal garbling of the function f distributed among the parties. As generating a garbled circuit does not involve the inputs of the parties, this phase will be input independent. In fact, we will describe an ideal functionality for every gate g in the circuit that parties will compute in parallel. We describe this ideal functionality next.

Each party P_i ($i \in [n]$) samples random keys $k^0_{w,i}, k^1_{w,i}$ and mask $\lambda_{w,i}$ for every wire $1 \leq i \leq W$ except for wires that carry inputs of party P_j for any $j \neq i$. Let g be a gate in the circuit with α, β as input wires and γ as output wire. Then all parties will engage in a multiparty protocol for gate g that will compute the following functionality:

P_i provides as input the masks $\lambda_{\alpha,i}, \lambda_{\beta,i}, \lambda_{\gamma,i}$, the keys chosen for the wire γ, i.e. $k^0_{\gamma,i}$ and $k^1_{\gamma,i}$. If either wire α or β carries the input of party P_j, then only P_j provides the mask bit for this wire. For simplicity, we will describe the functionality as a randomized algorithm. The actual functionality that the parties will compute is the deterministic variant of the functionality that additionally takes as input from the parties auxiliary randomness $r_{\mathsf{aux},i}$ which will be XORed and used by functionality as its random tape.

1. The functionality receives the inputs from all parties and computes the HIDDEN MASKS as $\lambda_\alpha = \bigoplus_{i=1}^n \lambda_{\alpha,i}$ $\lambda_\beta = \bigoplus_{i=1}^n \lambda_{\beta,i}$, and $\lambda_\gamma = \bigoplus_{i=1}^n \lambda_{\gamma,i}$.

[7]This can be formalized in the UC-framework where the environment selects the inputs of the parties. Here, necessarily the inputs have to be determined before the environment gets to see any communication (via the adversary) in the protocol.

2. The combined key for the wire γ is the concatenation of the keys contributed by all the parties, computed as

$$K^0_\gamma = (k^0_{\gamma,1}, \dots, k^0_{\gamma,n})$$
$$K^1_\gamma = (k^1_{\gamma,1}, \dots, k^1_{\gamma,n})$$

Then it computes the key that needs to be encrypted in each row corresponding to the garbling of gate g as follows: For $b, b' \in \{0,1\}$ row (b, b') contains $R^{bb'}_g$ computed as:

$$R^{bb'}_g = (K^{v \oplus \lambda_\gamma}_\gamma, v \oplus \lambda_\gamma) \text{ where } v = g(b \oplus \lambda_\alpha, b' \oplus \lambda_\beta)$$

3. Then the functionality secret shares (using XOR) each of the four values $(R^{00}_g, R^{01}_g, R^{10}_g, R^{11}_g)$ as $\{R^{00}_{g,i}\}^n_{i=1}, \{R^{01}_{g,i}\}^n_{i=1}, \{R^{10}_{g,i}\}^n_{i=1}$ and $\{R^{11}_{g,i}\}^n_{i=1}$ and party P_i receives as output from the functionality $(R^{00}_{g,i}, R^{01}_{g,i}, R^{10}_{g,i}, R^{11}_{g,i})$.

The functionality is formalized in Figure 3. All parties after receiving their share for each gate g, they output it locally as their local state for the preprocessing phase.

Online Phase In this phase, each party first broadcasts the VISIBLE MASKS corresponding to its input wires. Then the shares received in the preprocessing phase are encrypted with the keys it contributed along with keys on all input wires corresponding to the VISIBLE MASKS revealed by the respective party. This will allow all parties to recombine the shares and then evaluate the garbled circuit and obtain the VISIBLE MASKS of the output wires. The parties then reveal their contribution to the HIDDEN MASKS for the output wires which can be combined with VISIBLE MASKS to obtain the actual output.

1. In Round 1, each party P_i with input $x_i = (x_{i_1} \cdots x_{i_m})$ computes the VISIBLE MASKS corresponding to the wires carrying their input, which are wires $i_1, \dots, i_m$ according to our convention. It can compute this locally, as the HIDDEN MASKS for these wires were determined only by P_i and the VISIBLE MASKS is the XOR of the actual value on the wire and the HIDDEN MASKS. More precisely,

   Round 1: P_i broadcasts ("Input Wire i_j" , Λ_{i_j}) where $\Lambda_{i_j} = \lambda_{i_j} \oplus x_{i_j}$ for $j \in [m]$
   After all parties receive VISIBLE MASKS for every input wire, they broadcast encryptions of the shares received in the preprocessing phase and their contributions to the keys corresponding to the VISIBLE MASKS for every input wire, namely:

2. In Round 2, for every input wire w,

   Round 2: P_i broadcasts ("P_i's key for Input wire w" , $k^{\Lambda_w}_{w,i}$),

   All parties receive the keys corresponding to input wires w and locally compute the combined key K_w as follows

$$K_w = (k_{w,1}, \dots, k_{w,n})$$

   For every gate g with input wires α, β and output wire γ,

   Round 2: P_i broadcasts :

   ("Row (0,0)", $c^{00}_{g,i,\text{left}} = \mathsf{FEE.Enc}_{k^0_{\alpha,i}}(\text{params}, s^{00}_{g,i,\text{left}})$, $c^{00}_{g,i,\text{right}} = \mathsf{FEE.Enc}_{k^0_{\beta,i}}(\text{params}, s^{00}_{g,i,\text{right}})$)

   ("Row (0,1)", $c^{01}_{g,i,\text{left}} = \mathsf{FEE.Enc}_{k^0_{\alpha,i}}(\text{params}, s^{01}_{g,i,\text{left}})$, $c^{01}_{g,i,\text{right}} = \mathsf{FEE.Enc}_{k^1_{\beta,i}}(\text{params}, s^{01}_{g,i,\text{right}})$)

   ("Row (1,0)", $c^{10}_{g,i,\text{left}} = \mathsf{FEE.Enc}_{k^1_{\alpha,i}}(\text{params}, s^{10}_{g,i,\text{left}})$, $c^{10}_{g,i,\text{right}} = \mathsf{FEE.Enc}_{k^0_{\beta,i}}(\text{params}, s^{10}_{g,i,\text{right}})$)

   ("Row (1,1)", $c^{11}_{g,i,\text{left}} = \mathsf{FEE.Enc}_{k^1_{\alpha,i}}(\text{params}, s^{11}_{g,i,\text{left}})$, $c^{11}_{g,i,\text{right}} = \mathsf{FEE.Enc}_{k^1_{\beta,i}}(\text{params}, s^{11}_{g,i,\text{right}})$)

where $s^{bb'}_{g,i,\mathsf{left}}, s^{bb'}_{g,i,\mathsf{right}}$ is a random XOR sharing of $R^{bb'}_{g,i}$.

Evaluation: After receiving the messages from all parties, P_i evaluates the Garbled Circuit. Recall that for each input wire w of the circuit, the parties possess Λ_w provided by the party whose input the wire is carrying. Define $b_w = \Lambda_w$ for every input wire.

Now they carry out the evaluation on the topological order of the gates g as follows: The parties pick the ciphertexts for Row (b_α, b_β), namely $(c^{b_\alpha b_\beta}_{g,i,\mathsf{left}}, c^{b_\alpha b_\beta}_{g,i,\mathsf{right}})$ for every $i \in [n]$, and decrypts them using $k^{b_\alpha}_{\alpha,i}$ and $k^{b_\beta}_{\beta,i}$ that can be obtained from $K^{b_\alpha}_\alpha$ and $K^{b_\beta}_\beta$. The decryption will yield $s^{b_\alpha b_\beta}_{g,i,\mathsf{left}}$ and $s^{b_\alpha b_\beta}_{g,i,\mathsf{right}}$. They XOR all the shares to obtain $R^{b_\alpha b_\beta}_g$ which by our construction is

$$R^{b_\alpha b_\beta}_g = (K^{v \oplus \lambda_\gamma}_\gamma, v \oplus \lambda_\gamma)$$

where $v = g(\beta_\alpha \oplus \lambda_\alpha, b_\beta \oplus \lambda_\beta)$ where $\lambda_w = \bigoplus_{i=1}^n \lambda_{w,i}$. Define $b_\gamma = v \oplus \lambda_\gamma$ and continue the evaluation.

3. Finally, in Round 3, all parties obtain a key and mask for the output wire, namely, $(K^{b_W}_W, b_W)$. Then,

   Round 3: P_i broadcasts ("Output Wire W" , $\lambda_{W,i}$)
   The final output is then computed by all parties as $(\bigoplus_{i=1}^n \lambda_{W,i}) \oplus b_W$.

Correctness: Let bit_w is the actual value in the wire w when the circuit C is fed as input $x_1, \ldots, x_n$ We will show inductively that for every wire w, the parties will obtain

$$\text{Key } K^{\Lambda_w}_w \text{ and mask } \Lambda_w$$

where $\Lambda_w = \lambda_w \oplus \mathsf{bit}_w$. This will prove correctness because corresponding to the output wire W, the parties obtain $\{\lambda_{W,i}\}_{i \in [n]}$ with which they compute

$$(\bigoplus_{i=1}^n \lambda_{W,i}) \oplus \Lambda_W = \mathsf{bit}_W.$$

Hence, it suffices to demonstrate our induction hypothesis to prove correctness.

Base case: Input wires of circuit. For gate g with input wires α, β that correspond to the input of the circuit (i.e. carries the input of some party), by our construction the parties have $K^{\Lambda_\beta}_\alpha$ and $K^{\Lambda_\beta}_\beta$. This follows from the fact that for an input wire w carrying an input bit of P_i, P_i broadcasts $\lambda_w \oplus \mathsf{bit}_w$ in Round 1.

Induction step: Let g be an arbitrary gate such that the parties possess $(K^{\Lambda_\alpha}_\alpha, \Lambda_\alpha)$ and $(K^{\Lambda_\beta}_\beta, \Lambda_\beta)$. We will show that it can obtain $K^{\Lambda_\gamma}_\gamma, \Lambda_\gamma$ where $\Lambda_\gamma = \lambda_\gamma \oplus \mathsf{bit}_\gamma$. Recall that for this gate g, the parties will use Row $(\Lambda_\alpha, \Lambda_\beta)$ and decrypt $(c^{\Lambda_\alpha \Lambda_\beta}_{g,i,\mathsf{left}}, c^{\Lambda_\alpha \Lambda_\beta}_{g,i,\mathsf{right}})$ for every $i \in [n]$ using $k^{\Lambda_\alpha}_{\alpha,i}$ and $k^{\Lambda_\beta}_{\beta,i}$ respectively and these keys are contained in $K^{\Lambda_\alpha}_\alpha$ and $K^{\Lambda_\beta}_\beta$. The decryption will yield $s^{\Lambda_\alpha \Lambda_\beta}_{g,i,\mathsf{left}}$ and $s^{\Lambda_\alpha \Lambda_\beta}_{g,i,\mathsf{right}}$. The parties add the shares computed for $i \in [n]$ which according to our functionality from the preprocessing phase is $R^{\Lambda_\alpha, \Lambda_\beta}_g$. By our construction

$$R^{\Lambda_\alpha, \Lambda_\beta}_g = (K^{v \oplus \lambda_\gamma}_\gamma, v \oplus \lambda_\gamma) \text{ where } v = g(\Lambda_\alpha \oplus \lambda_\alpha, \Lambda_\beta \oplus \lambda_\beta).$$

By our induction hypothesis, we have that $\Lambda_\alpha \oplus \lambda_\alpha = \mathsf{bit}_\alpha$ and $\Lambda_\beta \oplus \lambda_\beta = \mathsf{bit}_\beta$. which implies that $v = g(\mathsf{bit}_\alpha, \mathsf{bit}_\beta) = \mathsf{bit}_\gamma$ and that the parties obtain

$$(K^{\mathsf{bit}_\gamma \oplus \lambda_\gamma}_\gamma, \mathsf{bit}_\gamma \oplus \lambda_\gamma) = (K^{\Lambda_\gamma}, \Lambda_\gamma).$$

This concludes the induction step and the proof of correctness.

MPC Functionality $\mathcal{F}^{g}_{\mathsf{share}}$

Let $1 \leq \alpha, \beta \leq W$ be the identities of the input wires of gate g and $1 \leq \gamma \leq W$ be the identity of the output wire of g.

- Party P_i provides as input to the functionality $\lambda_{\alpha,i}, \lambda_{\beta,i}, \lambda_{\gamma,i}$; $k^0_{\gamma,i}, k^1_{\gamma,i}$, and $r_{\mathsf{aux},i}$.
- Let $\lambda_\alpha = \bigoplus_{i=1}^n \lambda_{\alpha,i}$, $\lambda_\beta = \bigoplus_{i=1}^n \lambda_{\beta,i}$, and $\lambda_\gamma = \oplus_{i=1}^n \lambda_{\gamma,i}$ and

$$\begin{aligned}
\chi_1 &= \lambda_\gamma \oplus g(\lambda_\alpha, \lambda_\beta) & R^{00}_g &= K^0_\gamma \oplus \left(\chi_1 \wedge (K^1_\gamma \oplus K^0_\gamma)\right)\\
\chi_2 &= \lambda_\gamma \oplus g(\lambda_\alpha, 1 \oplus \lambda_\beta) & R^{01}_g &= K^0_\gamma \oplus \left(\chi_2 \wedge (K^1_\gamma \oplus K^0_\gamma)\right)\\
\chi_3 &= \lambda_\gamma \oplus g(1 \oplus \lambda_\alpha, \lambda_\beta) & R^{10}_g &= K^0_\gamma \oplus \left(\chi_3 \wedge (K^1_\gamma \oplus K^0_\gamma)\right)\\
\chi_4 &= \lambda_\gamma \oplus g(1 \oplus \lambda_\alpha, 1 \oplus \lambda_\beta) & R^{11}_g &= K^0_\gamma \oplus \left(\chi_4 \wedge (K^1_\gamma \oplus K^0_\gamma)\right)
\end{aligned}$$

  where $K^0_\gamma = (k^0_{\gamma,1}, \ldots, k^0_{\gamma,n})$ and $K^1_\gamma = (k^1_{\gamma,1}, \ldots, k^1_{\gamma,n})$. We use the

  $\oplus$ operator above to denote the XOR operation applied bitwise and $\chi_j \wedge (K^1_\gamma \oplus K^0_\gamma)$ for $j \in \{1, 2, 3, 4\}$ is interpreted as computing logical and operation of χ_j with every bit of $(K^1_c \oplus K^0_c)$.
- The functionality computes random XOR shares for $(R^{00}_g, R^{01}_g, R^{10}_g, R^{11}_g)$ as $\{R^{00}_{g,i}\}_{i=1}^n, \{R^{01}_{g,i}\}_{i=1}^n, \{R^{10}_{g,i}\}_{i=1}^n$ and $\{R^{11}_{g,i}\}_{i=1}^n$ (secret shares are generated using randomness $r_{\mathsf{aux}} = \bigoplus_{i=1}^n r_{\mathsf{aux},i}$). It sends $(R^{00}_{g,i}, R^{01}_{g,i}, R^{10}_{g,i}, R^{11}_{g,i})$ to party P_i for every $i \in [n]$.

Figure 3: The secret sharing functionality $\mathcal{F}^{g}_{\mathsf{share}}$ for gate g.

Simulation. Without loss of generality we can assume that the simulator learns the output of the computation in the very beginning, even if nobody is corrupted. This can be achieved by the standard transformation $f'((x_1, \mathsf{mask}_1), \ldots, (x_2, \mathsf{mask}_2)) = (f_1(x_1, \ldots, x_n) \oplus \mathsf{mask}_1, \ldots, f_n(x_1, \ldots, x_n) \oplus \mathsf{mask}_n)$. That is, we instruct parties to pick their random masks, run MPC protocol to compute the function f', xor i-th chunk of output with mask_i and learn the output $f_i(x_1, \ldots, x_n)$. The simulator can generate a random output in the beginning and later, upon corruption of party i, open its mask appropriately.

The simulation on a high-level will generate the encryptions under active keys using FEE.Enc and the rest of the rows using FEE.SimEnc. Just as in the two-party setting, we need to define the function that will be used to equivocate the difference ciphertexts. On a high-level, our simulation will proceed as follows: The function embedded in the ciphertexts will have two modes that can be activated by one of its inputs. In one mode, say $\mathsf{mode} = 0$, the function will be a constant function, outputting a hardwired constant const. In the second mode, $\mathsf{mode} = 1$, the function on input x will compute $C(x)$, figure the actual values $\mathsf{bit}_\alpha, \mathsf{bit}_\beta$ and bit_γ just as in the two-party setting and then output something so that the the shares from all decryptions under keys from all parties will add up to reveal the correct key. Upto $n-1$ corruptions, the ciphertexts will be revealed under $\mathsf{mode} = 0$, and when n^{th} party is corrupted and all inputs of parties become known, the ciphertext will be revealed under $\mathsf{mode} = 1$.

We will now describe the simulation's procedure in the Preprocessing Phase and Online Phase.

Preprocessing Phase: In this phase, the simulation will have to simulate the communication in the sub-protocol used to compute the functionality $\mathcal{F}^{g}_{\mathsf{share}}$ and any adaptive corruptions that occur in the middle of the execution of this sub-protocol. The simulation will rely on the standard adaptive simulation of the underlying GMW protocol used to realize this functionality. In order to carry out the simulation according the GMW protocol, when a party is corrupted, the simulation needs to provide its input and output functionality $\mathcal{F}^{g}_{\mathsf{share}}$. We describe next how to determine this.

- Upto $n-1$ corruptions, the simulator generates randomly chosen keys $k^0_{w,i}$ and $k^1_{w,i}$ and mask $\lambda_{w,i}$ for every wire w and sets P_1's input as it would be in an honest execution and the output received to be random. Namely, it sets $(R^{00}_{g,i}, R^{01}_{g,i}, R^{10}_{g,i}, R^{11}_{g,i})$ to be all random strings of the appropriate length.
- If the n^{th} party P_i is corrupted, before or at the end of the preprocessing phase, then the simulator firsts generates random keys $k^0_{w,i}$ and $k^1_{w,i}$ and mask $\lambda_{w,i}$ just as for the other $n-1$ corruptions. Then, to compute the outputs $(R^{00}_{g,i}, R^{01}_{g,i}, R^{10}_{g,i}, R^{11}_{g,i})$, it will first run an honest computation of $\mathcal{F}^g_{\mathsf{share}}$ using the inputs of all parties and compute the actual rows $(R^{00}_g, R^{01}_g, R^{10}_g, R^{11}_g)$ and then sets the share for P_i so that it adds up the actual row. Namely, it sets $R^{bb'}_{g,i} = (\oplus_{i \neq j} R^{bb'}_{g,j}) \oplus R^{bb'}_g$ for every g and $b, b' \in \{0,1\}$ which will be the output of P_i.

Online Phase: If all parties were corrupted in the Preprocessing Phase, the simulator learns all inputs $x_1, \ldots, x_n$ and runs Online Phase honestly. From now on we assume that at least one party remained uncorrupted, and therefore the simulation has to produce communication for the parties and address adaptive corruptions in this Phase. Recall that the parties in this phase first broadcast VISIBLE MASKS corresponding to the input wires. This is followed by the parties broadcasting encryptions and their contribution to the keys corresponding to the VISIBLE MASKS for the input wires of the circuit. Then they evaluate the Garbled Circuit and then broadcast their contribution to the HIDDEN MASKS masks for the output wire.

More formally, for all parties that have already been corrupted, the Simulation simple carries out the honest code with the inputs and outputs determined for these parties in the preprocessing phase. For the remaining parties. For the remaining uncorrupted parties, the Simulation proceeds as follows:

1. In Round 1, it samples a random key using the $\mathsf{FEE.Gen}$ function for every wire w and every party P_i that has not yet been corrupted, denoted by $k_{w,i}$. Next, it will determine an active path by sampling random bits for Λ_w for each wire w. This it will sample only for all intermediate wires and input wires of the circuit carrying inputs of uncorrupted parties. Then the active row in each gate g with input wires α and β is given by $(\Lambda_\alpha, \Lambda_\beta)$. Corresponding to each input wire w of the circuit that carries an input bit from an uncorrupted P_i, the simulator places in the transcript the following broadcast message from party P_i:

   Round 1: P_i broadcasts ("Input Wire i_j" , Λ_{i_j})
2. In Round 2, the parties broadcast their contribution to the keys for the input wires of the circuit and the ciphertexts for each row of each gate.

   For each input wire of the circuit, the simulator places on the transcript the following message for uncorrupted P_i's.

   Round 2: P_i broadcasts ("P_i's key for Input wire w " , $k_{w,i}$),

   For each gate g with input wires α, β and output wire γ, the simulator samples random strings $s^{bb'}_{g,i,\mathsf{left}}$ and $s^{bb'}_{g,i,\mathsf{right}}$ for $b, b' \in \{0,1\}$. It also runs $\mathsf{SimTrap}$ to generate trapdoors $\mathsf{td}^{\Lambda_\alpha(1\oplus\Lambda_\beta)}_{g,i,\mathsf{right}}, \mathsf{td}^{(1\oplus\Lambda_\alpha)\Lambda_\beta}_{g,i,\mathsf{left}}, \mathsf{td}^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)}_{g,i,\mathsf{left}}$ and $\mathsf{td}^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)}_{g,i,\mathsf{right}}$. Next, it places on the transcript:

   Round 2: P_i broadcasts :

$(Row(\Lambda_\alpha, \Lambda_\beta),$

$$\begin{aligned} c_{g,i,\mathsf{left}}^{\Lambda_\alpha\Lambda_\beta} &= \mathsf{FEE.Enc}_{k_{\alpha,i}}(\mathsf{params}, s_{g,i,\mathsf{left}}^{\Lambda_\alpha\Lambda_\beta}), \\ c_{g,i,\mathsf{right}}^{\Lambda_\alpha\Lambda_\beta} &= \mathsf{FEE.Enc}_{k_{\beta,i}}(\mathsf{params}, s_{g,i,\mathsf{right}}^{\Lambda_\alpha\Lambda_\beta})) \end{aligned}$$

$(Row(\Lambda_\alpha, 1 \oplus \Lambda_\beta),$

$$\begin{aligned} c_{g,i,\mathsf{left}}^{\Lambda_\alpha(1\oplus\Lambda_\beta)} &= \mathsf{FEE.Enc}_{k_{\alpha,i}}(\mathsf{params}, s_{g,i,\mathsf{left}}^{\Lambda_\alpha(1\oplus\Lambda_\beta)}), \\ c_{g,i,\mathsf{right}}^{\Lambda_\alpha(1\oplus\Lambda_\beta)} &= \mathsf{FEE.SimEnc}(F_{g,i}^{\Lambda_\alpha(1\oplus\Lambda_\beta)})) \end{aligned}$$

$(Row(1 \oplus \Lambda_\alpha, \Lambda_\beta),$

$$\begin{aligned} c_{g,i,\mathsf{left}}^{(1\oplus\Lambda_\alpha)\Lambda_\beta} &= \mathsf{FEE.SimEnc}(F_{g,i}^{(1\oplus\Lambda_\alpha)\Lambda_\beta}), \\ c_{g,i,\mathsf{right}}^{(1\oplus\Lambda_\alpha)\Lambda_\beta} &= \mathsf{FEE.Enc}_{k_{\beta,i}}(\mathsf{params}, s_{g,i,\mathsf{right}}^{(1\oplus\Lambda_\alpha)\Lambda_\beta})) \end{aligned}$$

$(Row(1 \oplus \Lambda_\alpha, 1 \oplus \Lambda_\beta),$

$$\begin{aligned} c_{g,i,\mathsf{left}}^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)} &= \mathsf{FEE.SimEnc}(F_{g,i,\mathsf{left}}^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)}), \\ c_{g,i,\mathsf{right}}^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)} &= \mathsf{FEE.SimEnc}(F_{g,i,\mathsf{right}}^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)})) \end{aligned}$$

Now, we define the functions $F_{g,i}^{\Lambda_\alpha(1\oplus\Lambda_\beta)}, F_{g,i}^{(1\oplus\Lambda_\alpha)\Lambda_\beta}, F_{g,i,\mathsf{left}}^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)}, F_{g,i,\mathsf{right}}^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)}$.
In fact, all these functions for party P_i will have a similar program. On a high-level all functions take as input $(x_1, \ldots, x_n, \mathsf{mode})$. If the $\mathsf{mode} = 0$, these functions output a fixed hardcoded string and if the $\mathsf{mode} = 1$, it outputs the "correct small key" by first computing the right bit assignment for the wire and then computing the other key $\overline{K}_\gamma$ for the output wire as follows: recall that this key is concatenation of small keys for each party. The program F_i will compute i-th chunk of this key by running Equiv on $(x_1, \ldots, x_n, \mathsf{mode} = 1)$, and all other chunks by running Equiv on $(0^n, \ldots, 0^n, \mathsf{mode} = 0)$[8]. Then in order to output this key $\overline{K}_\gamma$ that is a concatenation of the small keys, the simulator will mask with shares that will be the fixed value decrypted from the remaining ciphertexts when the $\mathsf{mode} = 0$ (and the fixed values that were encrypted by the keys that were not equivocated).
The basic idea is that upto $n-1$ corruptions, the keys are equivocated with input set as $(0^n, \ldots, 0^n, \mathsf{mode} = 0)$. If P_i is the last party to be corrupted, the simulator knows the input of all parties and equivocates P_i's keys using the input $(x_1, \ldots, x_n, \mathsf{mode} = 1)$. We present formal descriptions of our functions in Figures 4-6.

3. After broadcasting the encryptions, in Round 3, the simulator needs to reveal the $\lambda_{W,i}$ corresponding to the output wire. If not all parties have been corrupted, then it simply reveals $\lambda_{W,i}$ at random so that $(\bigoplus_{i=1}^n \lambda_{W,i}) \oplus \Lambda_W = C(x_1, \ldots, x_n)$, which is the required answer. Now we formally deal with corruptions in the online phase.
   - Upto $n-1$ corruptions: The simulator needs to produce the view of party P_i both in the Preprocessing and Online Phase. The active keys have already been sampled. The inactive keys $\widehat{k}_{w,i}$ are computed by equivocating them with input $(0^n, \ldots, 0^n, \mathsf{mode} = 0)$ with trapdoor $\mathsf{td}_{w,i}$. Then, the view in the Preprocessing Phase is computed by setting P_i's input with the two keys $(k_{w,i}, \widehat{k}_{w,i})$ for every input wire w carrying an input bit of P_i and output as random shares for each row of each gate.
     The random coins of encryption is generated by running the $\mathsf{FEE.Adapt}$ algorithm.
   - n^{th} corruption. When the n^{th} party is corrupted, the simulator again needs to generate the view of P_i in the Preprocessing and Online Phase. First, it generates the inactive

[8] The choice of inputs 0^n is arbitrary: recall that if $\mathsf{mode} = 0$, the program ignores the other part of input.

```
Program $F_{g,i}^{\Lambda_\alpha(1\oplus\Lambda_\beta)}(x_1,\ldots,x_n,\mathsf{mode})$

Constants: $C, g, K_\gamma, \{\mathsf{td}_{\gamma,j}\}_{j=1}^n, \{(s_{g,j,\mathsf{left}}^{\Lambda_\alpha(1\oplus\Lambda_\beta)}, s_{g,j,\mathsf{right}}^{\Lambda_\alpha(1\oplus\Lambda_\beta)})\}_{j=1}^n, \Lambda_\gamma$.
if $\mathsf{mode}=0$ then output $s_{g,i,\mathsf{right}}^{\Lambda_\alpha(1\oplus\Lambda_\beta)}$.
else if $\mathsf{mode}=1$
    1. Evaluate $C(x_1,\ldots,x_n)$ and learn bit assignments $\mathsf{bit}_\alpha, \mathsf{bit}_\beta$ of input wires $\alpha,\beta$ of gate $g$.
    2. For $j \neq i$, generate
            $\widehat{k}_{\gamma,j} \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_{\gamma,j}, (0^n,\ldots,0^n,\mathsf{mode}=0))$.
       Generate
            $\widehat{k}_{\gamma,i} \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_{\gamma,i}, (x_1,\ldots,x_n,\mathsf{mode}=1))$.
    3. Compute $\mathsf{mask} = (\bigoplus_{j=1}^n s_{g,j,\mathsf{left}}^{\Lambda_\alpha(1\oplus\Lambda_\beta)}) \oplus (\bigoplus_{j\neq i} s_{g,j,\mathsf{right}}^{\Lambda_\alpha(1\oplus\Lambda_\beta))})$ and Key $\overline{K}_\gamma = (\widehat{k}_{\gamma,1},\ldots,\widehat{k}_{\gamma,n})$.
    4. if $g(\mathsf{bit}_\alpha,\mathsf{bit}_\beta) = g(\mathsf{bit}_\alpha, 1\oplus\mathsf{bit}_\beta)$ then set $M_g^{\Lambda_\alpha,1\oplus\Lambda_\beta} = K_\gamma||\Lambda_\gamma$;
       else set $M_g^{\Lambda_\alpha,1\oplus\Lambda_\beta} = \overline{K}_\gamma||(1\oplus\Lambda_\gamma)$.
    5. Output $M_g^{\Lambda_\alpha,1\oplus\Lambda_\beta} \oplus \mathsf{mask}$.
end if
```

Figure 4: Program in Row $(\Lambda_\alpha, 1\oplus\Lambda_\beta)$ for SimEnc.

keys $\widehat{k}_{w,i}$ by equivocating on input $(x_1,\ldots,x_n,\mathsf{mode}=1)$ and trapdooor $\mathsf{td}_{w,i}$. Then, to compute the outputs $(R_{g,i}^{00}, R_{g,i}^{01}, R_{g,i}^{10}, R_{g,i}^{11})$, it will first run an honest computation using the inputs of all parties and compute the actual rows $(R_g^{00}, R_g^{01}, R_g^{10}, R_g^{11})$ and then sets the share for P_i so that it adds up the actual row. Namely, it sets $R_{g,i}^{bb'} = (\oplus_{i\neq j} R_{g,j}^{bb'}) \oplus R_g^{bb'}$ for every g and $b,b' \in \{0,1\}$.
The random coins of encryption are again computed by running the $\mathsf{FEE.Adapt}$.

Proof of security: We provide a brief proof sketch showing that our simulation is correct. Upto $n-1$ corruptions, it is easy to show that the joint view of all parties are indistinguishable. This follows from the fact that the inputs and outputs that set in the Preprocessing phase are identically distributed and indistinguishability of the view in this phase follows from the adaptive security of the underling GMW protocol. For the the online phase, as long as at most $n-1$ parties are corrupted, one share of each inactive key (and therefore the whole inactive key) is unknown. For the active row, the messages and randomness are identically distributed in the real and ideal world. To argue this formally, we can consider a sequence of hybrids where all encryptions are changed from being generated according to $(\mathsf{FEE.SimTrap}, \mathsf{FEE.SimEnc}, \mathsf{FEE.Adapt})$ to $(\mathsf{FEE.Gen}, \mathsf{FEE.Enc})$. We can use the standard security as we know the messages that are encrypted in each of these ciphertexts (they are simply random fixed strings).

When the n^{th} party is corrupted, we will consider a sequence of hybrids from the simulation to the real world as follows: We will follow a topological order on the gates and replace the encryptions from being encrypted using the FEE simulation to the honest encryption. Again, we crucially rely on the fact that we know all the messages to be encrypted and the fact that in this hybrid the reduction knows the inputs of all parties $(x_1,\ldots,x_n)$ even if all the parties are not corrupted (as we are proving security in the semi-honest setting where the inputs are determined at the beginning of the computation (by the environment) and independent of the views of the parties). Then we essentially follow the same set of hybrids as in the proof of Equivocal Garbling (with the difference that we secret share each key among $2n$ values instead of just 2). We remark that in this sequence we will continue to simulate the views

```
Program $F^{(1\oplus\Lambda_\alpha)\Lambda_\beta}_{g,i}(x_1,\ldots,x_n,\mathsf{mode})$

Constants: $C, g, K_\gamma, \{\mathsf{td}_{\gamma,j}\}_{j=1}^n, \{(s^{(1\oplus\Lambda_\alpha)\Lambda_\beta}_{g,j,\mathsf{left}}, s^{(1\oplus\Lambda_\alpha)\Lambda_\beta}_{g,j,\mathsf{right}})\}_{j=1}^n, \Lambda_\gamma$.
if $\mathsf{mode} = 0$ then output $s^{(1\oplus\Lambda_\alpha)\Lambda_\beta}_{g,i,\mathsf{left}}$.
else if $\mathsf{mode} = 1$
    1. Evaluate $C(x_1,\ldots,x_n)$ and learn bit assignments $\mathsf{bit}_\alpha, \mathsf{bit}_\beta$ of input wires $\alpha, \beta$ of gate $g$.
    2. For $j \neq i$, generate
           $\widehat{k}_{\gamma,j} \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_{\gamma,j}, (0^n,\ldots,0^n,\mathsf{mode}=0))$.
       Generate
           $\widehat{k}_{\gamma,i} \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_{\gamma,i}, (x_1,\ldots,x_n,\mathsf{mode}=1))$.
    3. Compute $\mathsf{mask} = (\bigoplus_{j\neq i} s^{(1\oplus\Lambda_\alpha)\Lambda_\beta}_{g,j,\mathsf{left}}) \oplus (\bigoplus_{j=1}^n s^{(1\oplus\Lambda_\alpha)\Lambda_\beta}_{g,j,\mathsf{right}})$ and Key $\overline{K}_\gamma = (\widehat{k}_{\gamma,1},\ldots,\widehat{k}_{\gamma,n})$.
    4. if $g(\mathsf{bit}_\alpha,\mathsf{bit}_\beta) = g(\mathsf{bit}_\alpha, 1\oplus\mathsf{bit}_\beta)$ then set $M_g^{1\oplus\Lambda_\alpha,\Lambda_\beta} = K_\gamma||\Lambda_\gamma$;
       else set $M_g^{1\oplus\Lambda_\alpha,\Lambda_\beta} = \overline{K}_\gamma||(1\oplus\Lambda_\gamma)$.
    5. Output $M_g^{1\oplus\Lambda_\alpha,\Lambda_\beta} \oplus \mathsf{mask}$.
end if
```

Figure 5: Program in Row $(1\oplus\Lambda_\alpha,\Lambda_\beta)$ for SimEnc.

```
Program $F^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)}_{g,i,\mathsf{left}}(x_1,\ldots,x_n,\mathsf{mode})$

Constants: $C, g, K_\gamma, \{\mathsf{td}_{\gamma,j}\}_{j=1}^n, \{(s^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)}_{g,j,\mathsf{left}}, s^{(1\oplus\Lambda_\alpha)\Lambda_\beta}_{g,j,\mathsf{right}})\}_{j=1}^n, \Lambda_\gamma$.
if $\mathsf{mode} = 0$ then output $s^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)}_{g,i,\mathsf{left}}$.
else if $\mathsf{mode} = 1$
    1. Evaluate $C(x_1,\ldots,x_n)$ and learn bit assignments $\mathsf{bit}_\alpha, \mathsf{bit}_\beta$ of input wires $\alpha, \beta$ of gate $g$.
    2. For $j \neq i$, generate
           $\widehat{k}_{\gamma,j} \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_{\gamma,j}, (0^n,\ldots,0^n,\mathsf{mode}=0))$.
       Generate
           $\widehat{k}_{\gamma,i} \leftarrow \mathsf{FEE.Equiv}(\mathsf{td}_{\gamma,i}, (x_1,\ldots,x_n,\mathsf{mode}=1))$.
    3. Compute $\mathsf{mask} = (\bigoplus_{j\neq i} s^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)}_{g,j,\mathsf{left}}) \oplus (\bigoplus_{j=1}^n s^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)}_{g,j,\mathsf{right}})$ and Key $\overline{K}_\gamma = (\widehat{k}_{\gamma,1},\ldots,\widehat{k}_{\gamma,n})$.
    4. if $g(\mathsf{bit}_\alpha,\mathsf{bit}_\beta) = g(\mathsf{bit}_\alpha, 1\oplus\mathsf{bit}_\beta)$ then set $M_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} = K_\gamma||\Lambda_\gamma$;
       else set $M_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} = \overline{K}_\gamma||(1\oplus\Lambda_\gamma)$.
    5. Output $M_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} \oplus \mathsf{mask}$.
end if

Program $F^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)}_{g,i,\mathsf{right}}(x_1,\ldots,x_n,\mathsf{mode})$

Output $s^{(1\oplus\Lambda_\alpha)(1\oplus\Lambda_\beta)}_{g,i,\mathsf{right}}$
```

Figure 6: Programs in Row $(1\oplus\Lambda_\alpha, 1\oplus\Lambda_\beta)$ for SimEnc.

in the Preprocessing Phase and finally after all the encryptions have been changed, we will invoke the simulation property of the underlying GMW protocol for the Preprocessing phase.

□

Remark 1. *We remark that this construction requires merely the existence of an adaptive honest-but-curious Oblivious Transfer protocol . Such protocols can be constructed based on any simulatable public-key encryption [DN00] (or even the weaker trapdoor simulatable public-key encryption [CDMW09]).*

6.1 Corollaries

In this section, we describe our results for designing adaptively secure protocol against byzantine (i.e. malicious) adversaries.

Corollary 5. *Let f be a function with n inputs. Assuming the existence of collision resistant hash-functions and dense cryptosystems, there exists a $O(1)$-round multiparty protocol to securely realize f against adaptive malicious corruption of all parties.*

Proof. We will obtain this corollary by applying two known results with Theorem 4. The first result allows to compile honest-but-curious protocols to byzantine protocols in the Uniform Reference String model, i.e. the model where all parties have access to a uniformly distributed random string sampled by a trusted party:

Theorem 6 ([CLOS02b]). *Suppose Π_f is a r-round n-party protocol to securely realize a n-party function f against adaptive honest-but-curious corruptions. Assuming the existence of simulatable public-key encryption and dense cryptosystems, there exists an $O(r)$-round protocol that realizes f against adaptive byzantine (malicious) corruptions in the Uniform Reference String model.*

In essence, the idea behind this compilation is to perform a GMW-style compilation by adding a coin-tossing protocol at the beginning of the protocol and following each message in the protocol with a zero-knowledge proof. Coin-tossing protocol can in turn be realized in the ideal-commitment hybrid. Canetti et al. [CLOS02b] show how to realize an ideal-commitment can be realized in the Uniform Reference String model based on dense-cryptosystems [SP92] and simulatable public-key encryption scheme and Canetti and Fischlin show to how to realize an adaptive zero-knowledge proof using the same assumptions [CF01].

The second result by Garg and Sahai [GS12] shows how the Uniform Reference String in the plain model (i.e. with non-concurrent security):

Theorem 7 (Theorem 2.[GS12]). *Assuming collision-resistant hash functions and dense cryptosystems, there exists a constant-round protocol that securely realizes the Uniform String Functionality against byzantine (malicious) corruptions.*

Combining Theorems 6 and 7 with Theorem 4, and noting that simulatable public-key encryptions can be based on dense cryptosystems, we obtain this corollary. □

For the stronger concurrent security, we show how to construct $O(1)$-round multiparty protocols in the Common Reference String model (i.e. the model where the reference string is sampled from an arbitrary but fixed distribution) and Uniform Reference String model (i.e. in the Universal Composability framework [Can01]). Namely, we obtain the following Corollary using Theorem 4 from our work and Theorem 9.8 from [CLOS02b].

Corollary 8. *Let f be a deterministic function with n inputs. Assuming the existence of simulatable public-key encryption schemes, there exists a $O(1)$-round multiparty protocol to realize f with UC-security against adaptive byzantine (malicious) corruption of all parties in the Common Reference String model. Additionally assuming the existence of dense cryptosystems, there exists a $O(1)$-round protocol achieving UC-security against adaptive (malicious) byzantine corruptions in the Uniform Reference String model.*

Finally, we mention that we can combine our Theorem 4 with the result of Dachman-Soled et al. [DMRV13] and Venkitasubramaniam [Ven14] to achieve the stronger UC-security in various setup models such as common reference string, (stateful) tamper proof hardware model, timing model and non-uniform simulation model and we leave it as future work to identify the precise constant in the round complexity and minimal assumptions.

7 Leakage-Resilient Two-party and Multiparty Protocols

In this section we slightly modify our protocol from the previous section to obtain one that provides some leakage resilience.

We will be working in the regime, where the parties participate in a leakage-free input-independent preprocessing phase and then in an online phase, the parties engage in a protocol with their inputs where the adversary can issue arbitrary leakage queries (with bounded leakage).

We will rely on the work of Bitansky, et al. [BCH12] that show that if a protocol admits *oblivious simulation* then the protocol is leakage resilient. Oblivious simuation, roughly speaking, requires there be $n+1$ simulators $\mathsf{Sim}_0, \ldots, \mathsf{Sim}_{n+1}$ that are initiated with some common randomness and then Sim_i for $1 \leq i \leq n$ is tasked to simulate an adaptive corruption of party P_i only using P_i's input and the shared randomness while Sim_0 is tasked with simulating the transcript of their interaction.

First, we briefly discuss the reason that our MPC protocol from the previous section will not admit an oblivious simulation. Recall that, the high-level idea was that simulating upto $n-1$ corruption, simply required providing random shares for "inactive" rows and all keys in the "active" rows. When the n^{th} party is corrupted, the inputs of all parties is used to equivocate the view of this n^{th} party. By definition, this simulation is not oblivious as the simulation of a party is contingent on whether it is the last party to be corrupted or not.

Now, we propose our modification that will allow for oblivious simulation after a leakage-free input-independent preprocessing phase. We will provide a multiparty protocol for any number of parties. To obtain a two party leakage-resilient secure computation we simply instantiate our multiparty protocol for two parties.

We will first extend our definition of Functionally Equivocal Encryption to have a "group" property. Namely, we will require that a group of parties jointly compute encryption and decryption and the simulation provides individual trapdoors to the member of the group that can equivocate a partial key locally.

First, we describe the syntax for Functionally Equivocal Group Encryption:

- **Key generation.** $\mathsf{Gen}(1^\lambda, 1^n; r_{\mathsf{Gen}})$ and outputs n keys, namely $K = (k_1, \ldots, k_n)$.
- **Encryption.** $\mathsf{Enc}_K(\mathsf{params}, m; r_{\mathsf{Enc}})$ interprets params as function description size $|f|$, input length n and output length l. It outputs an encryption of m with respect to parameters params using randomness r_{Enc} and key k.
- **Decryption.** $\mathsf{Dec}_K(c)$ decrypts ciphertext c using key k and outputs plaintext m.
- **Ciphertext simulation.** Simulating a ciphertext comprises of two algorithms $(\mathsf{SimTrap}, \mathsf{SimEnc})$ where $\mathsf{SimTrap}$ on input $(1^\lambda, 1^n; r_{\mathsf{td}})$ outputs the trapdoors $(\mathsf{td}_1, \ldots, \mathsf{td}_n)$ and SimEnc on input $(f, (\mathsf{td}_1, \ldots, \mathsf{td}_n); r_{\mathsf{Sim}})$ outputs a ciphertext c with respect to a function f.
- **Equivocation.** For each $1 \leq i \leq n$,$\mathsf{Equiv}(x_i, \mathsf{td}_i)$ uses the equivocation trapdoor td_i to generate a single fake key $k_{\mathsf{eq},i}$ so that each simulated ciphertext c_{eq}, which was generated with respect to some function f will decrypt to $f(x_1, \ldots, x_n)$ under $(k_{\mathsf{eq},1}, \ldots, k_{\mathsf{eq},n})$.

Definition 5. Functionally Equivocal Group Encryption) *A tuple of algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{SimTrap}, \mathsf{SimEnc}, \mathsf{Equiv}$ *is a functionally equivocal group encryption, if the following properties hold:*

- **Correctness.** *For any* $m \in M$ $\Pr[\mathsf{Dec}_k(\mathsf{Enc}_k(f, m; r)) = m : r \leftarrow \{0,1\}^{|r|}, k \leftarrow \mathsf{Gen}(1^\lambda, 1^n)] > 1 - \mathsf{negl}(\lambda)$.
- **t-functional security.** *For every* PPT *adversary* $\mathcal{A}$ *on input* 1^λ*, there exists a negligible function* $\nu(\cdot)$ *such that the probability that it wins the following game with challenger* $\mathcal{C}(1^\lambda)$ *is at most* $\frac{1}{2} + \nu(\lambda)$.
  1. *The adversary* $\mathcal{A}$ *sends* t *functions* $f_1, \ldots, f_t$ *(where each* f_i *maps* n *bits to* l_i *bits) and inputs* $x_1, \ldots, x_n \in \{0,1\}^n$ *to* $\mathcal{C}$*;*
  2. $\mathcal{C}$ *computes the messages* $\{m_i \leftarrow f_i(x_1, \ldots, x_n)\}_{i=1,\ldots,t}$.
  3. *Next it generates keys and ciphertexts in two different ways:*
     - $\mathcal{C}$ *samples random FEE key* $K = (k_1, \ldots, k_n)$ *using* $\mathsf{Gen}(1^\lambda, 1^n)$*. For* $1 \le i \le t$*, it computes*
       $$c_i \leftarrow \mathsf{Enc}_K(\mathsf{params}_i, m_i)$$
       where $\mathsf{params}_i \leftarrow (|f_i|, n, l_i)$.
     - $\mathcal{C}$ *computes ciphertexts using* SimEnc *as follows:*
       $$\begin{aligned} \mathsf{td} = (\mathsf{td}_1, \ldots, \mathsf{td}_n) &\leftarrow \mathsf{SimTrap}(1^\lambda, 1^n), \\ c_i &\leftarrow \mathsf{SimEnc}(f_i, \mathsf{td}) \;\; \forall\, 1 \le i \; \le t \end{aligned}$$
       Next it computes $k_{\mathsf{eq},i} \leftarrow \mathsf{Equiv}(x_i, \mathsf{td}_i)$.
  4. $\mathcal{C}$ *tosses a coin* b.
     - *If* $b = 0$*,* $\mathcal{C}$ *sends* $(K, c_1, \ldots, c_t)$ *to* $\mathcal{A}$.
     - *If* $b = 1$*,* $\mathcal{C}$ *sends* $((k_{\mathsf{eq},1}, \ldots, k_{\mathsf{eq},n}), c_{\mathsf{eq},1}, \ldots, c_{\mathsf{eq},t})$ *to* $\mathcal{A}$.
  5. *A outputs a bit* b'.

  A wins if $b = b'$.

The main differences in the definition of FEGE from FEE is that:

1. There are n keys that are generated. Looking ahead, there is one key for each party.
2. The Simulation produces n different trapdoors that each can be equivocated to a partial key "obliviously" (i.e. locally) just using the respective input x_i. Decrypting the simulated ciphertext under the key that is a concatenation of the partial keys results in the message $f(x_1, \ldots, x_n)$.
3. We no longer require the randomness to be equivocated. We deliberately removed this requirement because we need a scheme that admits "oblivious" simulation of the keys and it is unclear how to achieve this that includes the randomness. Looking ahead, we will erase this randomness at the time of encryption, i.e. a preprocessing phase and will therefore yield a leakage-resilient online phase.

We will first describe how we can use an FEGE to design a leakage resilient MPC and then show how to construct an FEGE from one-way functions.

7.1 Leakage-resilient MPC from FEGE

The leakage-resilent MPC will proceed in two phases: An input-independent and leak-free preprocessing phase and an online phase where the adversary can issue leak queries. On a high-level, in the preprocessing phase, the parties will jointly compute the Garbled Circuit using the group encryption. Recall that in the two-party setting, the Garbler G alone created the circuit using the FEE. Here we will essentially run the same procedure, with the exception that we will jointly compute all encryptions use group encryption. The parties then erase all the randomness used in this phase and only keep the keys generated for the input wires of the Garbled Circuit. In the online phase, depending on their input x_i, they simply broadcast, for each input wire, their key corresponding to the particular bit in x_i. Simulation on the other hand would generate the Garbled Circuit following the simulation of the equivocal garbling and then simply provide all the randomness used to build the simulated garbling and the individual trapdoors to each of the simulators. Then to obliviously simulate the corruption of a party, the simulator uses that party's input x_i and trapdoor td_i designated for that party and outputs $\mathsf{Equiv}(x_i, \mathsf{td}_i)$. Such a simulation in the online phase is oblivious by definition.

Slightly more formally, the leakage-resilient MPC protocol proceeds as follows:

Preprocessing Phase: We will describe the preprocessing phase as an MPC functionality that will provide the parties with outputs. This can easily be realized as an MPC protocol of a deterministic functionality (say GMW) where all parties contribute randomness that will produce the required output. The parties upon receiving the output will erase all randomness used in the computation.

The key generation function of the FEGE is run twice for each wire to generate $K_w^b = (k_{w,1}^b, \ldots, k_{w,n}^b)$ for $b \in \{0,1\}$ and each wire w. Furthermore, a hidden mask λ_w is generated for each wire. Then the garbling is done according to the equivocal garbling procedure described in Section 4. More precisely, for gate g with α, β as input wires and γ as output wire, the following ciphertexts are created: $c_{g,\mathsf{left}}^{b\oplus\lambda_\alpha, b'\oplus\lambda_\beta} = \mathsf{FEGE.Enc}_{K_\alpha^b}(s_{g,\mathsf{left}}^{bb'})$ and $c_{g,\mathsf{right}}^{b\oplus\lambda_\alpha, b'\oplus\lambda_\beta} = \mathsf{FEGE.Enc}_{K_\beta^{b'}}(s_{g,\mathsf{right}}^{bb'})$ where $s_{g,\mathsf{right}}^{bb'} \oplus s_{g,\mathsf{left}}^{bb'} = K_\gamma^{g(b,b')}$. Finally, one share of an XOR sharing of all the garbled rows is provided to each party. Corresponding to an input wire w carrying an input of party P_i the functionality will provide only P_i with the hidden mask λ_w. All parties will also be provided with an output translation table that maps the keys for the final output wire to the actual value.

Online Phase: In the online phase, each party P_i on input x_i will first broadcast $\Lambda_w = \lambda_w \oplus \mathsf{bit}_w$ for every input wire that will carry the input x_i where bit_w is that bit from x_i flowing in wire w. Next, all parties upon receiving Λ_w for all input wires will broadcast $k_{w,j}^{\Lambda_w}$ and also broadcasts their shares of the garbled rows. All parties reconstruct the entire garbled circuit by XORing the shares of the garbled rows and then using the input keys obtained for every input wire, will evaluate the circuit to obtain Λ_w for the final output key and using the output translational table obtain the final answer.

The simulation proceeds as follows:

Preprocessing Phase: The simulation will generate all ciphertexts according to the simulation presented in Section 4. First it will generate one key for every wire (the dummy key) $K_w = (k_{w,1}, \ldots, k_{w,n})$ using the $\mathsf{FEGE.Gen}$ function. Then, for gate g, it will produce.

$$\begin{array}{lllllll}
Row(\Lambda_\alpha, \Lambda_\beta), & c_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta} & = \mathsf{FEE.Enc}_{K_\alpha} & (\mathsf{params}, & & s_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta} & ; r_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta}), \\
 & c_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta} & = \mathsf{FEGE.Enc}_{K_\beta} & (\mathsf{params}, & & s_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta} & ; r_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta}) \\
Row(\Lambda_\alpha, 1\oplus\Lambda_\beta), & c_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} & = \mathsf{FEGE.Enc}_{K_\alpha} & (\mathsf{params}, & & s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} & ; r_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}), \\
 & c_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta} & = \mathsf{FEGE.SimEnc} & (F_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}] & & & ; r_{\mathsf{Sim},g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}) \\
Row(1\oplus\Lambda_\alpha, \Lambda_\beta), & c_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta} & = \mathsf{FEGE.SimEnc} & (F_g^{1\oplus\Lambda_\alpha,\Lambda_\beta}[s_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}] & & & ; r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}), \\
 & c_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta} & = \mathsf{FEGE.Enc}_{K_\beta} & (\mathsf{params}, & & s_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta} & ; r_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}) \\
Row(1\oplus\Lambda_\alpha, 1\oplus\Lambda_\beta), & c_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} & = \mathsf{FEGE.SimEnc} & (\mathsf{Const}_g[s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}] & & & ; r_{\mathsf{Sim},g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}), \\
 & c_{g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta} & = \mathsf{FEGE.SimEnc} & (F_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}[s_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}] & & & ; r_{\mathsf{Sim},g,\mathsf{right}}^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}).
\end{array}$$

where $s_{g,\mathsf{left}}^{\Lambda_\alpha,\Lambda_\beta} = (k_\gamma || \Lambda_\gamma) \oplus s_{g,\mathsf{right}}^{\Lambda_\alpha,\Lambda_\beta}$.

The functions embedded in these programs $F_{g,\mathsf{right}}^{\Lambda_\alpha,1\oplus\Lambda_\beta}, F_{g,\mathsf{left}}^{1\oplus\Lambda_\alpha,\Lambda_\beta}, \mathsf{Const}_g, F_g^{1\oplus\Lambda_\alpha,1\oplus\Lambda_\beta}$ will be essentially the same as the ones described in Figure 1 with the exception that it will receive as input

$x_1, \ldots, x_n$ and will have constants $\mathsf{td}_{\gamma,1}, \ldots, \mathsf{td}_{\gamma,n}$ with which it will generate the other key $\widehat{k}_\gamma$ by running $\mathsf{Equiv}(\mathsf{td}_i, x_i)$ for every $i \in [n]$ and concatenating them.

Then the simulations for each party P_i will receive only trapdoors corresponding to their key for all input wires. Namely, they will receive $\mathsf{td}_{w,i}$ for every input wire w. They will also have XOR sharings of all the garbled rows constructed above.

Online Phase: In the online phase, the obliviously simulate a corruption party P_i, the simulator using the parties input x_i and trapdoor $\mathsf{td}_{w,i}$ for every input wire w simply outputs the one normal key $k_{w,i}$ that it generated and the other key $\widehat{k}_{w,i} = \mathsf{Equiv}(\mathsf{td}_{w,i}, x_i)$.

The proof of security will essentially follows from the security definition of the underlying FEGE scheme. Since all the intermediate randomness has been removed corrupting upto $n-1$ parties will not reveal anything beyond the output and the active rows. Upon corrupting the n^{th} party, the view generated will essentially be the simulated view according to the equivocal garbling described in Section 4 with the exception that no randomness is revealed.

7.2 Constructing FEGE

It only remains to construct the FEGE scheme that will allow for group encryption and decryption and also simulating in a way that will allow to "obliviously" equivocate the individual keys. The construction presented in Section 3 can be easily adapted to the group setting. An encryption in the FEE scheme is simply a simulated (statically secure) garbling. In the FEGE, the encryption will continue to be a simulated (statically secure) garbling with the exception that the keys corresponding to input wires will be decomposed into n parts depending on which input wire belongs to which part according to the function description. Namely, the keys corresponding to input wires w that carry input x_i will be part of key k_i.

The simulation for the FEGE scheme on the other hand will be identical to the FEE scheme with the exception that it will decompose the trapdoors (that are the two keys corresponding to the input wires) again according to which input it will carry. td_i will contain all pairs of keys (k_w^0, k_w^1) that wire w carries some bit of the input x_i.

We point out here that it will be crucial that the randomness need not be equivocated (recall that there is no Adapt function for FEGE) because obliviously equivocating this randomness is not possible in our approach. In fact, this is the reason why we need an input-independent preprocessing phase in our construction of the leakage-resilient MPC protocol.

References

[BCH12] Nir Bitansky, Ran Canetti, and Shai Halevi. Leakage-tolerant interactive protocols. In *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*, pages 266–284, 2012.

[BDL14] Nir Bitansky, Dana Dachman-Soled, and Huijia Lin. Leakage-tolerant computation with input-independent preprocessing. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 146–163, 2014.

[BGJ+13] Elette Boyle, Sanjam Garg, Abhishek Jain, Yael Tauman Kalai, and Amit Sahai. Secure computation against adaptive auxiliary information. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 316–334, 2013.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, pages 134–153, 2012.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, pages 503–513, 1990.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[CCD88] D. Chaum, C. Crepeau, and I. Damgaard. Multi-party unconditionally secure protocols. In *STOC*, pages 11–19, 1988.

[CDMW09] Seung Geol Choi, Dana Dachman-Soled, Tal Malkin, and Hoeteck Wee. Simple, black-box constructions of adaptively secure protocols. In *TCC*, pages 387–402, 2009.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In *CRYPTO '01*, pages 19–40, 2001.

[CFGN96] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 639–648, 1996.

[CGP15] Ran Canetti, Shafi Goldwasser, and Oxana Poburinnaya. Adaptively secure two-party computation from indistinguishability obfuscation. In *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, pages 557–585, 2015.

[CLOS02a] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 494–503, 2002.

[CLOS02b] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *STOC*, pages 494–503, 2002.

[CP16] Ran Cohen and Chris Peikert. On adaptively secure multiparty computation with a short CRS. In *Security and Cryptography for Networks - 10th International Conference, SCN 2016, Amalfi, Italy, August 31 - September 2, 2016, Proceedings*, pages 129–146, 2016.

[DKR14] Dana Dachman-Soled, Jonathan Katz, and Vanishree Rao. Adaptively secure, universally composable, multi-party computation in constant rounds. *IACR Cryptology ePrint Archive*, 2014:858, 2014.

[DMRV13] Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Muthuramakrishnan Venkitasubramaniam. Adaptive and concurrent secure computation from new adaptive, non-malleable commitments. In *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part I*, pages 316–336, 2013.

[DN00] Ivan Damgård and Jesper Buus Nielsen. Improved non-committing encryption schemes based on a general complexity assumption. In *CRYPTO*, pages 432–450, 2000.

[GJS11] Sanjam Garg, Abhishek Jain, and Amit Sahai. Leakage-resilient zero knowledge. In *CRYPTO*, pages 297–315, 2011.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, pages 39–56, 2008.

[GL90] Shafi Goldwasser and Leonid A. Levin. Fair computation of general functions in presence of immoral majority. In *CRYPTO*, pages 77–93, 1990.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[GP14] Sanjam Garg and Antigoni Polychroniadou. Two-round adaptively secure MPC from indistinguishability obfuscation. *IACR Cryptology ePrint Archive*, 2014:844, 2014.

[GR10] Shafi Goldwasser and Guy N. Rothblum. Securing computation against continuous leakage. In *CRYPTO*, pages 59–79, 2010.

[GS12] Sanjam Garg and Amit Sahai. Adaptively secure multi-party computation with dishonest majority. In *CRYPTO*, pages 105–123, 2012.

[GWZ09] Juan A. Garay, Daniel Wichs, and Hong-Sheng Zhou. Somewhat non-committing encryption and efficient adaptively secure oblivious transfer. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, pages 505–523, 2009.

[HJO$^+$16] Brett Hemenway, Zahra Jafargholi, Rafail Ostrovsky, Alessandra Scafuro, and Daniel Wichs. Adaptively secure garbled circuits from one-way functions. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*, pages 149–178, 2016.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, pages 572–591, 2008.

[MNPS04a] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302, 2004.

[MNPS04b] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 287–302, 2004.

[MR04] Silvio Micali and Leonid Reyzin. Physically observable cryptography (extended abstract). In *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings*, pages 278–296, 2004.

[Nie02] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, pages 111–126, 2002.

[Rog91] P. Rogaway. The round complexity of secure protocols. In *PhD Thesis,*, page MIT, 1991.

[SP92] Alfredo De Santis and Giuseppe Persiano. Zero-knowledge proofs of knowledge without interaction (extended abstract). In *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*, pages 427–436, 1992.

[Ven14] Muthuramakrishnan Venkitasubramaniam. On adaptively secure protocols. In *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, pages 455–475, 2014.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computation. In *FOCS*, pages 160–164, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.