Multiuser Versions of Verifiable and Secure Computation

Ko Dokmai, Saba Eskandarian

1 Introduction

We have discussed a number of schemes in class which are designed around providing various secure functionalities in the setting where a weak client wants to take advantage of the powers made available by the advent of cloud computing without giving up the privacy afforded by performing computation locally. In most of the schemes studied, there was one client who interacted with one powerful cloud server to accomplish some desired task. Two specific tools in this problem domain are the related but not identical goals of secure computation and verifiable computation. For each of these problems, we will explore a scheme which provides an extension of the desired functionality to a multi-client setting, where there are many weak clients who wish to work with a powerful cloud server in order to carry out some computation(s).

The importance of these multiparty variations on verifiable and secure computing is that they effectively capture some very common use cases in the world of cloud computing. The realization of secure instantiations of these tools could open the door to a best of both worlds scenario, where users can take advantage of the vast potential for collaboration and productivity offered by web technologies on the vanguard of innovation without the cost of sacrificing privacy for functionality. The two schemes covered here offer an early effort in that direction. The rest of this draft will consider the problems of Multi-user Verifiable Computation and "On-the-fly" Multiparty Computation. Each solution will be preceded with a brief presentation of several building blocks. The Multiparty Computation problem relies heavily on the construction of a Multikey Fully Homomorphic Encryption scheme, which will be given more attention.

Although the two schemes address slightly different problems in somewhat different settings, they are united thematically by the effort to extend existing cryptographic tools and primitives applicable to cloud computing into the multi-user regime. Both constructions we will see involve some notion of verifiability of the outsourced computation. Moreover, in both cases, the definition of security attained seems to be slightly less satisfying than the best one could imagine. In both casses, the ideal security definition could be used to create Virtual Black-Box Obfuscation, implying that, like VBB obfuscation, those stronger definitions are unattainable.

2 Multi-client Verifiable Computation

We will now consider the problem of Multi-client Verifiable Computation. The typical setting for verifiable computation involves a weak client and a powerful cloud server. The client wishes to delegate work to the server, but it wants to have the power to make sure that the results computed by the server are in fact correct. It must be able to do this, of course, with less work than it takes the server to compute the result in the first place, or the delegation would be useless. In the multi-client setting, the main difference arises from the fact that we have multiple clients who have

some function $f(x_i, ..., x_n)$ which they would like for the server to compute, and each client has an input x_i.

After formalizing this setting, we will consider a scheme for achieving multi-client verifiable computation first presented by Choi et al in [1]. The construction uses proxy oblivious transfer, garbled circuits, and fully-homomorphic encryption. It will only deliver the output of the function which all the clients wish to compute to one of the clients, but this can easily be resolved by having each client run the protocol in parallel. This gives the clients the added benefit that they can each compute a different function on the group's data. An additional property of this scheme is that it is non-interactive. This means that there is no additional communication between the server and clients other than the initial messages sent to the server and the final response delivered back to the first client.

2.1 Defining Multi-client Verifiable Computation

A number of definitions exist for multi-client verifiable computation, but all of them attempt to capture the essence of the setting informally described above: multiple parties $P_1...P_n$ wish to delegate the computation of function $f(x_1...x_n)$ to a cloud server, and each P_i has input x_i. The clients and the cloud engage in a protocol which results in the clients learning the result of the function from the cloud in a verifiable way and the cloud doing significantly more work than the clients. The clients only send and receive information from the cloud, not from each other (this reflects the real-world setting where many clients connect to a central cloud without any peer-to-peer communication). Since we will focus most of our attention on the construction of Choi et al ([1]), the definition presented here is closest to theirs. The most powerful definition of MVC, however, is found in [2]. That definition is more general than the Choi et al definition and more intuitively captures the nature of the interaction between the clients and the server. Interestingly, it is shown in [2], by a reduction from virtual black box obfuscation, that it is not possible to achieve MVC better than that described by their definition.

We define an n-party MVC scheme as a tuple of algorithms (KeyGen_1, KeyGen, EnFunc, EnInput_1, EnInput_i, Compute, Verify) with the following properties[1]. It is assumed that there is some public-key infrastructure available that allows all the parties to share their public keys with each other.

- $\text{KeyGen}_1(1^k) \rightarrow (pk_i, sk_i)$: This is run by the first client to obtain a key pair.
- $\text{KeyGen}(1^k, i) \rightarrow (pk_i, sk_i)$: This is run by each client P_2 through P_n to obtain a key pair. pk without a subscript refers to the vector containing all the pk_is (including that of P_1).
- $\text{EnFunc}(1^k, f) \rightarrow (\phi, \xi)$: This algorithm is run by P_1 who will eventually get the output of the function. ϕ is an encoded version of the function f that is sent to the server and ξ is the decoding secret kept by P_1 for use later.
- $\text{EnInput}_i(pk, sk_i, x_i) \rightarrow \chi_i$: Run by each client P_2 through P_n, this algorithm creates an encoded input from the client's input, client's secret key, and the set of public keys.
- $\text{EnInput}_1(pk, sk_1, \xi, x_1) \rightarrow (\chi_1, \tau)$: A special form of EnInput run by P_1, this algorithm additionally takes in the function secret and additionally outputs the input decoding secret,

[1] the EnInputs, Compute, and Verify can be run repeatedly for each call to the function that the clients make. The inputs/outputs labeling each function call with a different id are omitted here for simplicity.

τ, which is kept by P_1. Otherwise, it has the same inputs and outputs as the previous EnInput for all other clients.

- Compute$(pk, \phi, \chi) \rightarrow \omega$: This algorithm, run by the server after receiving inputs from all clients, generates an encoded output ω from the public keys, the encoded function, and the encoded inputs.

- Verify$(\xi, \tau, \omega) \rightarrow z$ or $\perp$: This is run by P_1 upon receiving the function output from the server. Given the function secret, the input secret, and the encoded output, P_1 either outputs the result of $f(x_1, ..., x_n)$ or $\perp$, indicating that the server tried to cheat.

In order for the functionality to be useful, it is necessary that the algorithms (except for the initial setup) run in less time than it would take for the clients to compute the function themselves. The completeness requirement is the scheme allows for the server to computer a function and give the clients an output which can be verified. The security requirement is that the server can't convince the clients to accept an incorrect output. The notion of security is formalized in [1], but this informal description will suffice for our purposes.

There are also two measures of privacy that need to be satisfied by a successful MVC scheme. The first is that no client learn any other client's inputs. The second is that the server not learn any client's inputs. That is, different inputs should be indistinguishable to the server.

Security. Although the scheme presented here is due to Choi et al in [1], it is claimed in [2] that the construction also meets a more stringent definition of security than that claimed by Choi et al. The security definition provided in [2] follows a slightly different model from most of the definitions we have seen so far. Most security definitions we have seen so far define a game where an adversary wins if certain conditions are met. In contrast, the definition of security used here is simulation based. This means that an ideal functionality where users do not need to use cryptographic tools is described (one in which we have trusted third parties, secure channels, etc) and then it is shown that the outputs of the parties in the real functionality are actually indistinguishable from the outputs of the parties in the ideal functionality.

The ideal functionality for MVC is not as straightforward as might be imagined. In fact, seemingly contrary to what is described above, the ideal functionality must be different based on whether or not the server is corrupted. A simple line of reasoning can give the intuition behind this need. If the server is not corrupted, all the clients send information to the server, the server computes the function and returns it to a client, who verifies it. If, however, the server is corrupted maliciously, the server could collude with one client, taking all the inputs from the other clients and giving the malicious client black-box access to the function with all the other clients' inputs fixed. It is shown in [2] that any scheme which could obviate this shortcoming would allow for the creation of Virtual Black Box obfuscation, which is known to be impossible. Thus, even in the ideal model of a secure MVC scheme, this attack will be possible. This means that although our protocol can be secure against cheating clients or a cheating server, there is no way to be secure against both at the same time. For our purposes, however, we will only consider the case where all parties are *semi-honest*, which is to say that they follow the rules of the protocol but they are curious to see if they can, without breaking the rules, learn some of the private information of the other parties.

In words, the ideal model for us is one where the users send their data to a trusted party who, at the server's request computes the desired function on the data. If the server is not corrupted, the trusted party simply returns the output to the first client. If the server is corrupted, the "trusted"

party provides black box access to the function with other clients' inputs filled in to any corrupted clients.

2.2 MVC Construction

2.2.1 Building Blocks

This construction uses three primitives to realize an MVC scheme: Proxy Oblivious Transfer, Garbled Circuits, and Fully Homomorphic encryption. A brief description of the definition of each is given below in preparation for the actual construction.

Proxy Oblivious Transfer. Proxy oblivious transfer is a variation of the well-known cryptographic primitive Oblivious Transfer (OT). The setting for OT is one where there is a sender who has two messages and a receiver who wants to receive a message from the sender. The sender wants to send one of the messages to the receiver (doesn't care which) and the receiver wants to read a message from the sender without the sender knowing which message was read. The ideal functionality of OT resembles a situation where the sender sends two messages to a trusted third party and the recever sends a bit b indicating which bit is to be received. The trusted party then sends the appropriate message to the receiver. It is quite a remarkable feat that such seemingly contradictory requirements can be met without a trusted intermediary.

In contrast, a proxy oblivious transfer protocol (POT) has 3 participants: A sender, a chooser, and a receiver (also known as the proxy). The sender sends two inputs m_0 and m_1 to the scheme, and the chooser sends a bit b. The receiver learns m_b and not b or m_{1-b}. The scheme consists of algorithms (SetupS, SetupC, Snd, Chs, Prx). The two setup functions create key pairs for the sender and chooser, which are given to the Snd and Chs functions along with the messages to be sent and the choosing bit. Snd and Chs output α and β, respectively, which are sent along with the public keys of the sender and chooser, to the receiver, who runs Prx. Prx returns y, which should correspond to m_b as described above. The security definition for proxy oblivious transfer is also in the real/ideal model. The ideal model is one where the sender sends m_0 and m_1 to a trusted party and the chooser sends the bit b to the trusted party. The trusted party then sends the message m_b to the receiver.

Garbled Circuits. A garbled circuit is a tool from secure computation that takes a circuit and creates a "garbled" version of it such that a server evaluating the circuit gets the result of computing the circuit, but does not learn the function being computed by the circuit. The inputs to the circuit are also garbled before running the circuit. The scheme consists of algorithms (Garble, Genc, Gdec):

- Garble$(1^k, C) \rightarrow pk, sk, \Gamma$: takes the security parameter and a circuit and outputs a key pair and the garbled circuit.
- Genc$(pk, x) \rightarrow c$: takes the public key and an input and outputs a garbled input.
- Geval$(\Gamma, c) \rightarrow Y$: Evaluates the garbled circuit on a garbled input, producing a garbled output.
- Gdec$(sk, c) \rightarrow y$: decrypts the garbled output. The correctness property is that $C(x) = y$.

A garbled circuit scheme that will be useful for our purposes has some additional requirements. First, the scheme must be projective, which means that if many inputs are all concatenated together

bit by bit, the final input $c' = c_1||c_2||...||c_n$ will be a valid input. The other requirement is for the additional security property that a server given input x can only correctly compute $f(x)$.

Fully Homomorphic Encryption. Fully Homomorphic encryption (FHE) allows for a server to carry out operations on encrypted data without learning anything about the data itself. An FHE scheme consists of (Fgen, Fenc, Fdec, Feval) where the first three algorithms are a typical public-key encryption scheme and Feval is a function which takes a public key, a circuit C, and a tuple of ciphertexts encrypted under the public key and outputs one ciphertext which is the encryption under the public key of C being run with the unencrypted versions of the input ciphertexts as input.

2.2.2 Construction

We will arrive at the construction by starting with a description of a single-use, single-client verifiable computation scheme and generalize it twice to get the multi-use multi-client verifiable computation scheme we want. Each generalization will use one more of the building blocks mentioned in the previous section.

The basic idea of the single-client, single-use scheme is that the client garbles the circuit representing the function f to be computed to the server. She then sends her garbled input. The server computes the circuit and sends back Y. The client decrypts the garbled Y to get y, the result of the delegated computation. To make this scheme able to handle multiple evaluations of the function, the only change is that the inputs are encrypted via an FHE after being garbled and the circuit is computed on the encrypted garbled inputs.

The problem that arises in the multi-client setting is that the secret keys for garbling inputs and for encrypting the garbled inputs are in the hands of the first client, P_1, and therefore inaccessible to the other clients. The solution to this problem lies in using proxy oblivious transfer. In all instances of POT used, P_1 is the sender, P_i is the chooser, and the server is the receiver. For each other client and each bit of the client's input, P_1 sends both the garbled and encrypted values of the bit if the bit were a 1 or a 0. P_i sends the appropriate bit of the input as the choosing bit. This means that the server learns the bits of the garbled and encrypted inputs, but does not learn any other bits or the raw inputs of the various clients. These bits can be assembled into the proper input for the garbled circuit and evaluated as before. Below is the description of (KeyGen_1, KeyGen, EnFunc, EnInput_i, EnInput_1, Computer, Verify) for this scheme:

- $\text{KeyGen}_1(1^k) \rightarrow (pk_i, sk_i)$: This is run by the first client in the setup phase. The client runs $\text{SetupS}(1^k)$.
- $\text{KeyGen}(1^k, i) \rightarrow (pk_i, sk_i)$: This is run by all clients except P_1. The client runs $\text{SetupC}(1^k)$.
- $\text{EnFunc}(1^k, f) \rightarrow (\phi, \xi)$: P_1 runs $\text{Garble}(1^k, f)$ to get (pk, sk, Γ). We set $\phi = \Gamma$ and $\xi = (pk, sk)$.
- $\text{EnInput}_1(pk, sk_i, x_i) \rightarrow (\chi_1, \tau)$: P_1 first runs $\text{Fgen}(1^k)$ to get an FHE key pair (pk_{FHE}, sk_{FHE}). For each bit index j in the input for each client $P_2...P_n$, P_1 computes $\text{Genc}(sk, 0)$ as well as $\text{Genc}(sk, 1)$, and encrypts both with the FHE to get x_{ij0} and x_{ij1}. It then computes $\text{snd}(pk_i, sk_1, x_{ij0}, x_{ij1})$ for each bit in each input. A total of $(n-1) * l$ (l is the length of the input) instances of POT are used. Additionally, produce the FHE-encryption of input x_1. All the preceding information is part of χ_1, which is sent to the server. τ is sk_{FHE} and is kept by P_1.

- EnInput$_i(pk, sk_i, x_i) \rightarrow \chi_i$: For each bit j from 1 to l , P_i sends Chs(pk_1, sk_i, x_{ij}).
- Compute$(pk, \phi, \chi) \rightarrow \omega$: This is run by the server after getting its inputs from the clients. It runs Prx() on all the inputs Snd/Chs inputs it received from the clients and reconstructs the encrypted garbled inputs into a ciphertext c. It then runs Geval$(\phi, c) = y$, which it sends to P_1 as ω. Note that y is still encrypted with the FHE.
- Verify$(\xi, \tau, \omega) \rightarrow z$ or $\perp$: P_1 runs Fdec$(\tau, \omega) = Z$ and then Gdec$(sk, Z) = z$ to get the final result of the delegated computation.

3 On-the-fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption

3.1 Introduction

Let us consider the setting where we have a computationally powerful but untrusted server—the "cloud"—and multiple computationally weak clients who wish to have their encrypted data stored and computed on the cloud by any inputted functions on any subset of data. We call this setting *on-the-fly multiparty computation* with the following properties: 1) The clients are unaware of the identity or even the number of other clients in the system. Additional data maybe later uploaded to the cloud by each client. 2) The cloud decides to evaluate dynamically chosen functions, which may be given by some clients or an outsider or are a part of the cloud service itself, on the data of arbitrary subset of clients chosen on the fly. The computation of the functions are non-interactive (i.e. no further communications with clients) and attains encrypted output. 3) The cloud and the subsets of clients chosen by the jointly retroactively approve the choice of functions and decrypt the output. This setting, if practical, can be extremely useful in real-life scenarios where users want to delegate their computations to a powerful cloud while still maintaining secrecy of the data and flexibility of the functions.

To approach a construction of this setting, let us imagine a special kind of fully homomorphic encryption schemes (FHE) called *multikey* FHE; in this multikey FHE scheme, any functions can be evaluated on ciphertexts which may be encrypted using many independent public keys; the resulting ciphertexts can be decrypted using a combination of relevant independent secret keys. To utilize this notion in our construction, we let the clients generate their own public and secret key pair and send their encrypted data using the generated public key to the server; the server then evaluates *offline* a given function on a subset of the ciphertexts (as required by the function) and output the resulting ciphertexts; finally, all the relevant clients partake in the decryption phase, where the decryption is done *securely*, i.e. secret keys are not revealed to other clients and the server, and *online* under a secure multi-party computation protocol (MPC). Few extra proofs between the clients and server are added to ensure that no participants deviate from the protocol. The construction of the multikey FHE and the protocol is showed possible by Lopez-Alt, Tromer, and Vaikuntanathan [3].

A question may arise whether we can achieve a similar protocol without the *online* decryption phase, which requires final interactions between the server and the clients. This is a useful notion in practice since in many applications, such as e-voting, it cannot be expected that all relevant users will become available to finalize the protocol. Unfortunately, Lopez-Alt, Tromer, and

Vaikuntanathan [3] shows a negative result that this notion implies virtual black-box obfuscation, which is known to be impossible.

3.2 Building Blocks

We shall begin this section by giving an overview of the protocol and showing where these building blocks fit in; then we shall describe each building block in more details.

Lopez-Alt, Tromer, and Vaikuntanathan in [3] shows that a construction of on-the-fly multiparty computation can be achieved using the following building blocks: multikey FHE based on NTRU, a server-aided secure MPC scheme, a non-interactive zero-knowledge (NIZK) argument system, a succinct non-interactive arguments of knowledge (SNARK) system. In the semi-malicious setting (i.e. the parties are assumed to follow the protocol precisely except for when random elements are sampled, and may try to learn from the information they have), multikey FHE is used by the clients to encrypt their data using independent encryption keys and by the server to evaluate the functions; the semi-malicious server-aided secure MPC scheme is used to securely decrypt the resulting ciphertexts with the restriction that all clients communicate directly with the server and not with other clients during the decryption. In the malicious setting (i.e. no parties are assumed to follow the protocol and may try in any way to learn others' secret), we turn the semi-malicious protocol into a malicious one as follows: The clients use NIZK argument system to prove to the server that their ciphertexts are well-formed, i.e. the public-secret key pair is consistent and the ciphertexts are encrypted using the correct key; the server uses SNARK system to prove correctness of the function evaluation to the clients; semi-malicious MPC protocol is replaced with malicious MPC protocol, with an extra check before decryption that the secret keys involved are correct.

Now we shall describe briefly all the above-mentioned building blocks.

NIZK. Non-interactive zero-knowledge proofs are defined similarly to interactive zero-knowledge proofs except that a *chanllenge* to the prover is not sent directly from the verifier, but generated by the prover from a common reference string (CRS) whose distribution is not controlled by the prover; thus no interaction is necessary between the prover and the verifier.

Fait and Shamir [4] showed how to convert the Σ protocol (a zero-knowledge proof of knowledge protocol) into a NIZK argument in the random oracle model: Let c be a *challege*; we define $c := H(a, x)$ where H is the random oracle, x is the statement the prover tries to prove, and a is a message generated by the prover from x and a witness w in the Σ protocol. Note that both a and x are known to the prover, so the prover does not need any extra information from the verifier.

SNARK. Succinct Non-Interactive Arguments of Knowledge (SNARK) is a non-interactive proof-of-knowledge argument which requires that the length of the proof and the time required for its verification are polylogarthmic in the size of the witness and there exists an extractor that "extracts" a valid witness from any valid proof.

The construction of SNARKs uses Fait and Shamir's transformation [4] to transform Killian's *interactive* PCP system to a *non-interactive* one in the random oracle model. For more information, please refer to [6].

Server-Aided MPC. In this variation of secure multiparty computation, we require the setting where there exist a powerful server and multiple clients who want to securely evaluate some function, with the restriction that most of the computation is carried out by the server and that

the clients only communicate with the server, not each other. The security of server-aided MPC follows from the security of MPC that, informally, the parties involved in the computation cannot learn each other's input except for what the output of the function implies. The possibility of server-aided MPC protocols is explored by Komara, Mohassel, and Raykova [7].

Multikey FHE. An *N-key fully homomorphic encryption scheme* is the same as a regular FHE scheme with two changes. First, the homomorphic evaluation algorithm takes in polynomially many ciphertexts encrypted under at most N keys, together with the corresponding evaluation keys, and produces a ciphertext. Second, in order to decrypt the resulting ciphertext, one uses all the involved secret keys. A construction of multikey FHE from NTRU as shown by Lopez-Alt, Tromer, and Vaikuntanathan [3] is provided in the next section.

3.3 Multikey Fully Homomorphic Encryption from NTRU

The description of the multikey FHE from NTRU in this section is taken verbatim from [3]. We start by describing NTRU encryption, one of the earliest lattice-based cryptosystems. Then, we show how NTRU can be made fully homomorphic on a single key and then multiple independent keys.

NTRU Encryption. The scheme is parametrized by the ring $R \stackrel{\text{def}}{=} \mathbb{Z}[x]/\langle x^n+1\rangle$ (i.e. polynomials of degree n with integer coefficients), where n is a power of two, an odd prime number q and a $B-$bounded distribution χ over R, for $B \ll q$. By "B-bounded", we mean that the magnitude of the coefficients of a polynomial sampled from χ is guaranteed to be less than B. We define $R_q \stackrel{\text{def}}{=} R/qR$ (i.e. polynomials of degree n with integer coefficients modulo q), and use $[\cdot]_q$ to denote coefficient-wise reduction modulo q into the set $\left\{-\left\lfloor \frac{q}{2} \right\rfloor, \ldots, \left\lfloor \frac{q}{2} \right\rfloor\right\}$.

- $\mathsf{Keygen}(1^{\mathcal{K}})$: Key generation samples "small" polynomials $f', g \leftarrow \chi$ and sets $f \stackrel{\text{def}}{=} 2f' + 1$ so that $f \pmod 2 = 1$. If f is not invertible in R_q, it resamples f'. Otherwise, it computes the inverse f^{-1} of f in R_q and sets

$$\mathsf{sk} = f \quad \text{and} \quad \mathsf{pk} = [2gf^{-1}]_q$$

- $\mathsf{Enc}(\mathsf{pk}, m)$: To encrypt a bit $m \in \{0, 1\}$, the encryption algorithm samples "small" polynomials $s, e \leftarrow \chi$, and outputs the ciphertext

$$c = [hs + 2e + m]_q$$

- $\mathsf{Dec}(\mathsf{sk}, c)$: To decrypt a ciphertext c, the decryption algorithm computes $\mu = [fc]_q$ and returns $\mu \pmod 2$

Multikey Homomorphism. We now briefly describe the (multikey) homomorphic properties of the scheme and the challenges encountered when converting it into a fully homomorphic encryption scheme.

Let $c_1 = [h_1 s_1 + e_1 + m_1]_q$ and $c_2 = [h_2 s_2 + e_2 + m_2]$ be ciphertexts under two different keys $h_1 = [2g_1 f_1^{-1}]_q$ and $h_2 = [2g_2 f_2^{-1}]_q$, respectively. We claim that $c_{\mathsf{add}} \stackrel{\text{def}}{=} [c_1 + c_2]_q$ and $c_{\mathsf{mult}} \stackrel{\text{def}}{=} [c_1 c_2]_q$

decrypt to $m_1 + m_2$ and $m_1 m_2$ respectively, under the joint secret key $f_1 f_2$. Indeed, notice that

$$\begin{aligned} f_1 f_2 &= 2(f_1 f_2 e_1 + f_1 f_2 e_2 + f_2 g_1 s_1 + f_1 g_2 s_2) + f_1 f_2 (m_1 + m_2) \\ &= 2e_{\mathsf{add}} + f_1 f_2 (m_1 + m_2) \end{aligned}$$

for a slightly larger noise element e_{add}. Similarly,

$$\begin{aligned} f_1 f_2 (c_1 c_2) = 2(&2g_1 g_2 s_1 s_2 + g_1 s_1 f_2 (2e_2 + m_2) + g_2 s_2 f_1 (2e_1 + m_1) + \\ &f_1 f_2 (e_1 m_2 + e_2 m_1 + 2e_1 e_2)) + f_1 f_2 (m_1 m_2)) \\ = 2&e_{\mathsf{mult}} + f_1 f_2 (m_1 m_2) \end{aligned}$$

for a slightly larger noise element e_{mult}. This shows that the ciphertexts $c_{\mathsf{add}} \stackrel{\mathsf{def}}{=} [c_1 + c_2]_q$ and $c_{\mathsf{mult}} \stackrel{\mathsf{def}}{=} [c_1 c_2]_q$ can be correctly decrypted to the sum and the product of the underlying messages, respectively, as long as the error does not grow too large.

Extending this to circuits, we notice that the secret key required to decypt a ciphertext c that is the output of a homomorphic evaluation on ciphertexts encrypted under N different keys, is $\prod_{i=1}^{N} f_i^{d_1}$, where d_i is the degree of the ith variable in the polynomial function computed by the circuit. Thus, decrypting a ciphertext that was the product of a homomorphic evaluation requires knowing the circuit! This is unacceptable even for somewhat homomorphic encryption.

We employ the relinearization technique to essentially reduce the degree from d_i to 1, so that the key needed to decrypt the evaluated ciphertext is now $\prod_{i=1}^{N} f_i$. This guarantees that decryption is dependent on the number of keys N but independent of the circuit computed. After using relinearization, we can show that the resulting scheme is multikey somewhat homomorphic.

From Multikey Somewhat to Fully Homomorphic Encryption. Once we obtain a multikey somewhat homomorphic encryption scheme, we can apply known techniques to convert it to a multikey fully homomorphic encryption scheme. In particular, we use modulus reduction to increase the circuit depth that the scheme can handle in homomorphic evaluation. This yields a leveled homomorphic scheme where the efficiency of the algorithms depends on the number of independent keys and the circuit depth. Finally, using Gentry's bootstrapping theorem [8] for multikey setting, we can convert the leveled homomorphic scheme into a fully homomorphic scheme, in which the algorithms are independent of the circuit depth.

3.3.1 Semi-malicious Protocol

Offline Phase: The clients samples independent key pairs $(\mathsf{pk}_i, \mathsf{sk}_i, \mathsf{ek}_i)$, encrypt their input under their corresponding public key: $c_i \leftarrow \mathsf{Enc}(\mathsf{pk_i}, x_i)$, and send this ciphertext to the server along with the public and evaluation keys $(\mathsf{pk}_i, \mathsf{ek}_i)$.

Online Phase: Once a function has been chosen, together wit ha corresponding subset of computing parties V:

Step 1: The server performs the *multikey homomorphic evaluation* of the desired circuit on the corresponding ciphertexts, and broadcasts the evaluated ciphertexts to all computing parties (i.e. all parties in V).

Step 2: The computing parties (i.e. parties in V) run the server-aided generic MPC protocol to decypt the evaluated ciphertext using their individual secret keys sk_i.

It is important to emphasize again that in the server-aided MPC protocol in Step 2, the clients only communicate with the server and not other clients.

3.3.2 Malicious Protocol

The protocol in section 1.3.1 can be shown to be secure against semi-malicious adversaries. We then turn to show how to modify the protocol to achieve security against malicious adversaries. We make three modifications, described below.

Modifying the Decryption Protocol. The first modification we make is to change the decryption protocol in Step 2 of the online phase to check that the secret key being used is a valid secret key for the corresponding public and evaluation keys. This ensures that if decryption is successful, then in particular, a corrupted party knows a valid secret key $\widetilde{\mathsf{sk}}_i$. This secret key binds the corrupted party to the input $\widetilde{x}_i = \mathsf{Dec}\left(\widetilde{\mathsf{sk}}_i, \widetilde{c}_i\right)$, which by semantic security of the FHE, must be independent of the honest inputs.

Adding Zero-Knowledge Proofs. We further require that in the offline phase, each party create a non-interactive zero-knowledge (NIZK) proof π_i^{ENC} showing that the ciphertext c_i is well-formed (i.e. that there exists plaintext x_i and randomness s_i such that $c_i = \mathsf{Enc}(\mathsf{pk}_i, x_i; s_i)$. This guarantees that for a corrupted party, $\mathsf{Dec}\left(\widetilde{\mathsf{sk}}_i, \widetilde{c}_i\right) \neq \perp$ and thus the party really "knows" an input $\widetilde{x}_i$. Furthermore, it guarantees that the ciphertexts c_i are fresh encryptions, which is important in our setting of fully homomorphic encryption where we must ensure that the error stays low in a homomorphic evaluation.

Adding Verification of Computation. Finally we must rely on a succinct argument system to ensure that the server performs homomorphic computation correctly. Due to the dynamic nature of our on-the-fly model, we are unable to use verifiable computation protocols in the pre-processing model or succinct argument with a reference string that depends on the circuit being computed. These would require the clients to perform some pre-computation dependent on the circuit to be compute before knowing the circuit, or to interact with the server after a function has been selected and compute in the time proportional to the circuit-size of the function.

When the total size of the inputs is small enough to be broadcasted to all parties, it suffices for the server to use any succinct arguments to prove that it carries out the computation correctly as specified. However, in the case when the total size of the inputs is too large, the clients need to also send the hash of their input to the server as a part of the proof of knowledge.

References

[1] S.G. Choi, J. Katz, R. Kumaresan, C. Cid, *Multi-client non-interactive verifiable computation*, in TCC, pages 499-518, 2013.

[2] S. Gordon, J. Katz, F. Liu, E. Shi, H. Zhou, *Multi-Client Verifiable Computation with Stronger Security Guarantees*, in Theory of Cryptography, pages 144-168, 2015.

[3] A. Lopez-Alt, E. Tromer, V. Vaikuntanathan, *On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption*, in STOC, H. J. Karloff and T. Pitassi, eds., ACM, New York, 2012, pp. 1219-1234.

[4] A. Fiat, A. Shamir, *How to prove yourself: Practical solutions to identification and signature problems*, in Andrew M. Odlyzko, editor, CRYPTO, volume 264 of Lecture Notes in Computer Science, pages 186-194, Springer, 1986.

[5] J. Kilian, *Improved efficient arguments (preliminary version)*, in Don Coppersmith, editor, CRYPTO, volume 963 of Lecture Notes in Computer Science, pages 311-324, Springer, 1995.

[6] P. Valiant, *Incrementally verifiable computation or proofs of knowledge imply time/space efficiency*, in Ran Canetti, editor, TCC, volume 4948 of Lecture Notes in Computer Science, pages 118, Springer, 2008.

[7] S. Kamara, P. Mohassel, M. Raykova, *Outsourcing multi-party computation*, Cryptology ePrint Archive, Report 2011/272, 2011, `http://eprint.iacr.org/`

[8] C. Gentry, *A fully homomorphic encryption scheme*, PhD thesis, Stanford University, 2009, `crypto.stanford.edu/craig` .