How SCRAM Implements MPC using TFHE

Landon Chu `lschu@mit.edu`

Lucas Novak `ldnovak@mit.edu`

Kristin Sheridan `kms20@mit.edu`

Nicolas Zhang `nxyzhang@mit.edu`

6.857 Spring 2020 Final Project

Contents

1 Introduction

In the field of medical research, functions computed on large amounts of patient data are often useful for many purposes, such as determining potential risk factors for disease or the efficacy of a particular medication. However, medical records are highly confidential and, in many cases, a particular hospital does not have enough patient data to find trends that are statistically significant. A system of secure multiparty computation (MPC) can help solve this problem by allowing multiple parties to compute arbitrary functions on all of their private inputs without revealing information about the inputs (beyond the result of the function evaluation). This kind of system is highly useful not only in the medical field, but across many industries in which multiple parties may want to know the answer to some communal function but the needed information is confidential or comprises trade secrets.

In this paper, we will look at the foundations of a particular MPC system that uses Threshold Fully Homomorphic Encryption (THFE) and the theory behind what makes this system work. We will also look at a server-based implementation of this system currently being built by MIT's Internet Policy Research Initiative (IPRI) and the potential pitfalls of such a system, as well as the risks of MPC in general. We also propose some practical measures that can be taken to help alleviate some of the risk present in using this system.

2 Background

2.1 Fully Homomorphic Encryption

Fully homomorphic encryption refers to a kind of encryption in which some computation on ciphertexts produces an encryption of the desired computation on the plaintexts. Because addition and multiplication are a complete basis for any circuit, an FHE scheme supporting addition and multiplication support computation of any circuit on ciphertexts. Construction of such a scheme is possible under the Learning With Errors (LWE) assumption.

Definition 2.1. *[1] The LWE assumption states that for a vector* a *sampled uniformly at random from* $\mathcal{R}_q = \mathbb{Z}_q[x]/f(x)$ *for integers* q, n*, if* b *is sampled uniformly at random from* $\mathbb{Z}_q$*, the pair* (a, b) *is polynomial-*

time indistinguishable from $(a, a \cdot s + e)$, where s is sampled from distribution φ, and e is sampled from distribution χ, where χ is Gaussian with small standard deviation and φ is uniform over $\mathcal{R}_q$.

In order to use an FHE scheme for MPC, Asharov, Jain, and Wichs specifically look at FHE schemes in which keys are homomorphic as well. This means that the addition of two ciphertexts encrypted with different secret keys leads to an encryption of the sum of the plaintexts with a secret key that is the sum of the original secret keys [1]. In order to ensure that the ciphertexts resulting from encryption under the new secret key are secure, the new secret key should be indistinguishable from random, even if the second key is chosen by an adversary. Asharov, Jain, and Wichs show that such a scheme is possible when the adversary's choice of key is slightly limited.

Asharov *et al* specifically address leveled fully homomorphic encryption, which is FHE in which circuits up to a given depth D can be evaluated in the scheme (rather than circuits of arbitrary depth). The scheme presented, based on [2], requires a different secret key for each level of the circuit in order to ensure encryption of products doesn't get too large. Additionally, the algorithm to evaluate a function f on the plaintexts may be different than an algorithm performing f on the ciphertexts, but the decryption of the result should produce $f(x_1, ..., x_N)$, where the x_i are the plaintexts.

2.2 Threshold Fully Homomorphic Encryption

In the previous section, we described FHE schemes, which are symmetric or public key encryption schemes in which separate algorithms generate keys, encrypt ciphertexts, decrypt ciphertexts, and evaluate circuits on the ciphertexts. In threshold fully homomorphic encryption, or THFE, key generation and decryption are protocols between N parties, not algorithms performed by a single party. Decryption requires knowledge of the secret key, so it must still be done through a communication protocol between individual parties, each of which only has a share of the secret key. In the construction given by Asharov *et al*, circuit evaluation can also be done by a single party without knowledge of the secret key, which means the work required for evaluation can be outsourced to a single party or to an outside computer [1].

2.3 MPC via TFHE

2.3.1 General case

Using a secure TFHE scheme, it's possible to construct an MPC scheme allowing multiple parties to compute a function on their distributed inputs without revealing any other information about the inputs they have. Asharov *et al* construct such a scheme in which the parties first compute the two rounds of their THFE scheme. At the end of the first round, they all have the public key and their share of the secret key. In the second round, they all encrypt their plaintexts and broadcast them so that anyone with the public evaluation key can compute the circuit on the ciphertexts. Finally, a third round is added in which each party can decrypt the result and broadcast the decrypted result of the circuit on the combined inputs.

Because the number of needed secret keys depends on the circuit, the communication complexity of this protocol is not circuit-independent, but rather linearly dependent on the depth of the circuit being computed. However, not all parties are required to compute the circuit, so a complicated circuit with limited depth can still be outsourced to a third party computer.

The authors additionally show that this scheme is secure in the presence of a "semi-malicious" adversary corrupting any number of parties, where "semi-malicious" refers to parties that are honest except in their choice of random coins. A semi-malicious adversary may choose their random coins selectively, rather than truly at random. They additionally show that if communication complexity is increased by requiring all parties to give a zero-knowledge proof that they are following the protocol honestly in each round (up to a manipulation of the random coins), this scheme is secure against malicious attackers as well.

2.3.2 The specific construction of MIT SCRAM

The Secure Cyber Risk Aggregation and Measurement (SCRAM) system, currently under development by MIT's Internet Policy Research Initiative (IPRI) is a server-based implementation of the MPC protocol presented in [1]. The main cryptographic primitive that was used for SCRAM was the scheme of Brakerski, Fan, and Vercuteran [3, 4], from here on referred to as the BFV scheme. Homomorphic operations are defined in this scheme for ciphertexts decryptable with the same secret key. The IPRI group's approach to designing an MPC algorithm based on the BFV scheme and following the work of [1] was to design a key generation protocol and corresponding decryption protocol where all parties participating in the MPC protocol could encrypt their data with the same public key, but a ciphertext could only be decrypted if authorization is given by all parties. The algorithms described below provide more detail of their specific instantiation of the framework of [1].

2.3.3 Distributed Key Generation

Consider the standard BFV key generation protocol, which outputs a public key secret key pair of the form

$$(\mathsf{pk}, \mathsf{sk}) = \big((a, a \cdot s + e),\ s\big)$$

where a is a uniformly random sample over the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/f(x)$ where $f(x)$ is some degree-n polynomial. The key generation protocol relies on the common random string (CRS) model and begins with the assumption that all participants have access to the same truly random seed for a pseudorandom number generator (PRNG). All participants then use this seed to generate the same pseudorandom sample a from $\mathcal{R}_q$. Each party i then generates the following key pair:

$$(\mathsf{pk}_i, \mathsf{sk}_i) = \big((a, a \cdot s_i + e_i),\ s_i\big)$$

Once all these public keys are generated, the shared public key is computed by summing the second component of each $\mathsf{pk}_i = (a, \mathsf{pk}_i[1])$ over $\mathcal{R}_q$ to get the following:

$$\mathsf{pk} = \big(a,\ \sum_i \mathsf{pk}_i[1]\big) = \big(a,\ a \cdot \sum_i s_i + \sum_i e_i\big)$$

The result of this sum is a well-formed public key for the BFV scheme, where the corresponding secret key $\mathsf{sk} = \sum_i s_i$ is already distributed in additive shares among the participants. In the next section, we describe how decryption is performed on a ciphertext encrypted with this public key.

2.3.4 Distributed Decryption

Consider a ciphertext in the standard BFV scheme, which has the following form:

$$\mathsf{ct} = \big(a,\ a \cdot s + \Delta m + e\big)$$

where Δ is a public scaling factor to prevent the small error term e from corrupting the message m. Decryption is performed by computing the following function on $\mathsf{ct} = (\mathsf{ct}[0], \mathsf{ct}[1])$.

$$m = \Big\lfloor \frac{\mathsf{ct}[1] - s \cdot \mathsf{ct}[0]}{\Delta} \Big\rceil = \Big\lfloor \frac{\Delta m + e}{\Delta} \Big\rceil$$

The IPRI group's distributed decryption protocol collaboratively computes the numerator from "decryption shares" generated by each party, then the division and flooring is computed in the clear. Recall from the previous section that the secret key has the form $s = \sum_{i=1}^{k} s_i$, and each party i has a share s_i of the secret key. We assume that all parties know the total number of parties k as well as the ciphertext $\mathsf{ct} = (a,\ a \cdot s + \Delta m + e)$ that is to be decrypted. Each client i takes their share of the secret key s_i and generates the following decryption share:

$$d_i = a \cdot s + \Delta m + e - k \cdot (a \cdot s_i + e'_i)$$

where all operations are over $\mathcal{R}_q$ and e'_i is an error term sampled from a discrete Gaussian with large standard deviation[1]. The additive term $(a \cdot s_i + e')$ is multiplied by k because when these shares are summed there are k terms with $a \cdot s = a \sum_i s_i$, each with a component $a \cdot s_i$ that must be subtracted away.

Each sample d_i is then published. The security of this step follows from the fact that the tuple $(a, a \cdot s_i + e'_i)$ is a well-formed Ring-LWE sample, which is computationally indistinguishable from (a, u) for a uniformly sampled $u \leftarrow \mathcal{R}_q$.

When all of the decryption shares have been published, the numerator in the decryption equation can be computed as follows:

$$\begin{aligned} d = \sum_{i=1}^{k} d_i &= k \cdot a \cdot s + k \cdot \Delta m + k \cdot e - \sum_{i=1}^{k} (k \cdot a \cdot s_i + e'_i) \\ &= k \cdot a \cdot s + k \cdot \Delta m + k \cdot e - k \cdot a \sum_{i=1}^{k} (s_i) - e' \\ &= k \cdot a \cdot s + k \cdot \Delta m + k \cdot e - k \cdot a \cdot s - e' \\ &= k \cdot \Delta m + k \cdot e - e' \end{aligned}$$

Note that the message m is multiplied by k in this numerator.

Once this numerator is computed, the message m can be recovered by simply dividing by $k\Delta$ and flooring.

$$m = \left\lfloor \frac{d}{k\Delta} \right\rfloor = \left\lfloor \frac{k \cdot \Delta m + k \cdot e - e'}{k\Delta} \right\rfloor$$

Correctness holds as long as the magnitude of the error term $ke - e'$ remains less than $k\Delta/2$.

Using these two protocols, we have an additive homomorphic encryption scheme that allows us to safely encrypt data from multiple parties under with the same BFV public key. This, in turn, allows us to use additive homomorphic operations on the encrypted data to compute the desired function of the MPC protocol, then use the above distributed decryption protocol to allow the parties to obtain the function output.

2.4 Security Guarantees

Let's compare here the model we are in (Ring Learning with Errors, or RLWE) to real life. For this we are first starting to explore how RLWE was implemented in MIT SCRAM.

[1]Such noise terms are sometimes referred to as "flooding" terms.

2.4.1 Notation

For a set S, the notation $a \overset{\$}{\leftarrow} S$ denotes sampling an element uniformly from S. For a distribution $\mathcal{D}$, the notation $b \overset{\$}{\leftarrow} \mathcal{D}$ denotes sampling an element according to the distribution.

2.4.2 Ring Learning with Errors

The security of the crypto primitives in the MIT SCRAM library is derived from the hardness of the Learning with Errors problem over polynomial rings. The ring in question will always be

$$\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$$

which, in words, is the ring of polynomials mod $x^n + 1$ with coefficients in $\mathbb{Z}_q$. In addition, n will always be a power of two . The modulus q will have additional structure covered in later sections, but for now just note that it is not prime, and, in fact, we will include its factorization in the public parameters.

Let's first state the definition of the problem, then unpack the definition into it's relevant components.

Definition 2.2 (Ring Learning with Errors)**.** *For integers n and q, a ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$, and an error distribution χ over $\mathcal{R}_q$, the decisional Ring Learning with Errors problem $\mathsf{RLWE}_{n,q,\chi}$ is to distinguish between samples of the form $(a, as + e)$ and (a, u) where $a, u \overset{\$}{\leftarrow} \mathcal{R}_q$ and $s, e \overset{\$}{\leftarrow} \chi$.*

Put another way, for parameters for which RLWE is hard, we have

$$(a, as + e) \approx_c (a, u)$$

2.4.3 Discrete Gaussian Distribution

The first question that will likely arise is what the error distribution χ is. For all the MIT SCRAM implementations we consider, χ is defined to be a discrete, zero-centered Gaussian mod q. Each coefficient of a sample $e \overset{\$}{\leftarrow} \chi$ is sampled independently, so only the 1-D distribution needs to be implemented. This is the first of two non-trivial algorithms that must be implemented, but of the two it is definitely the easier. In general, the discrete Gaussian is a somewhat tricky distribution to implement, since it is **not** equivalent to a rounded continuous Gaussian. However, here we are saved by the fact that the standard deviation σ of this Gaussian is small . This means that we can just precompute the probability of sampling each integer in the range of $[-5\sigma, 5\sigma]$ and then just sample from this finite set according to these probabilities.

2.4.4 Number Theoretic Transform

This is the second non-trivial algorithm that is pretty essential to implement for any code handling RLWE instances. The number theoretic transform (NTT) is essentially just the FFT over $\mathcal{R}_p$ for a prime p. Note that since the main modulus q is not prime, MIT IPRI didn't run an NTT over this modulus. This will be explained in the next section.

Putting aside the modulus issue for now, it should be clear that polynomial multiplication requires some form of FFT in order to achieve any level of practical performance. The degrees of the polynomials in question are at minimum 2048 and often much higher, so the naïve $O(n^2)$ multiplication would be pretty terrible (at least 100x slowdown). In addition, this is essentially the only operation that is not linear in n, so taking this step from $O(n^2)$ to $O(n \log(n))$ directly improves the bottleneck operation.

There are a number of ways to implement the NTT, but since the degree n is always a power of two we can use Cooley-Turkey style implementations to achieve $n \log(n)$ complexity. The algorithm in the code MIT IPRI shared does this. It also implements a precomputation step where the roots of unity are preprocessed once for a fast online time.

To connect the NTT over a prime to MIT IPRI's composite modulus q, they turned to the residue number system (RNS), also known as the Chinese Remainder Theorem (CRT) representation.

Stepping back for a bit, the high level issue that arises with RLWE schemes is that the modulus q must be very large. In particular, it is often much larger than the standard machine word. In order to avoid multi-precision integer operations, one can make use of the Chinese Remainder Theorem, which says that for primes p_i such that

$$\prod_{i=1}^{\ell} p_i = q$$

the following rings are isomorphic:

$$\mathbb{Z}_q \cong \mathbb{Z}_{p_1} \otimes \mathbb{Z}_{p_2} \otimes \ldots \otimes \mathbb{Z}_{p_\ell}$$

This means that an integer $x \in \mathbb{Z}_q$ can be represented as a vector of ℓ integers $(x_1, \ldots, x_\ell)$ where x_i is an element in $\mathbb{Z}_{p_i}$. This isomorphism allows us to take elements x in $\mathbb{Z}_q$ large modulus q and split them into ℓ elements where each of the smaller elements fits into a standard machine word. The bijection has nice linearity properties that allows it to commute with the NTT, so we can take a polynomial in $\mathcal{R}_q$, split it coefficient-wise into ℓ polynomials over the rings $\mathcal{R}_{p_1}, \ldots, \mathcal{R}_{p_\ell}$, perform and NTT over each of the rings, then recombine to get the evaluation representation of the original polynomial in $\mathcal{R}_q$.

2.5 Potential Weaknesses

For some functions, any MPC protocol that exactly computes $f(x_1, x_2, ..., x_N)$ is insecure given an adversary that knows all but one party's input. For example, in the case where f computes the average of its inputs, if the output and all inputs but one x_k are known, x_k can be computed, as f is just a linear function in x_k. If the adversary can additionally control the inputs of the other parties prior to computation, they can determine the exact input of party k for even more functions. For example, if f outputs the maximum of its inputs and all inputs are known to be non-negative, the adversary could set the inputs of all other parties to be 0 so that f outputs party k's value. These concerns are especially relevant in the case that there are only two parties, in which case either party will be able to exactly determine the other party's input for any bijective function f.

To address concerns of privacy when an adversary may control multiple parties, techniques used in maintaining differential privacy when releasing aggregate statistics may be used. For example, if N parties want to compute the average number of students that cheated on the test, we can let f compute the average over all inputs and assume all inputs are 0 or 1. In a model for differential privacy, all parties input their honest answers to an algorithm, A, that flips the bit comprising each input with probability $\frac{1}{3}$ before computing the final output [5]. A similar model could be used for MPC computation of the same question, but rather than a single trusted algorithm changing each input with some probability, each party must toss random coins to determine whether or not to input their correct answer. In this case, while the adversary controlling all other parties can determine the exact input by party k, it does not know if k's answer was honest or not. It does know that k reports honestly with probability $\frac{2}{3}$, so it does gain some information about k's input, but not as much as it would in the original MPC model.

Differential privacy models often intentionally add noise to all inputs, resulting in noise in the final output. Thus, using a noise-adding model may help obscure a party's input in the case that the inputs of the other parties are compromised, but it also decreases accuracy of the final output. Thus, adding this additional step may not be useful in all cases. In particular, if all parties are fairly confident that the other parties have not shared their inputs (which may be a more reasonable assumption with a larger number of participants) or if it is difficult to retrieve a particular x_k given $f(x_1, x_2, ..., x_N)$ and all other inputs, the extra distortion and effort may not be worth it. It's also important to note that these schemes do not completely obscure x_k and generally just add noise, so a party should still be cautious about using multi-party computation if they are concerned about another party getting *any* information about their input.

3 Evaluating SCRAM

MIT's Internet Policy Research Initiative (IPRI) is building a multiparty computation system called Secure Cyber Risk Aggregation and Measurement (SCRAM). SCRAM is using a central-server model based on TFHE where the server evaluates an operation on the participating parties' ciphertexts. This section will describe the goal state of the system and the security policy/properties of a fully implemented SCRAM.

3.1 Overview

Under SCRAM's MPC implementation, there is a central server which connects multiple parties together, facilitates distributed key generation, receives encrypted data from each party, performs operations on the encrypted data, and sends the encrypted aggregate results back to each party. It is worth noting that the MPC protocol doesn't theoretically require there to be a central server; we discuss the benefits and potential security tradeoffs of this decision in section 3.3.

Currently, SCRAM is more developed at a theoretical than at a practical level. The low level encryption algorithms have been built in C++, and the server is written in Django; parties connect to the server via a browser page written in JavaScript. There is a demo of the current implementation but, because it is an early demo of system capabilities, there are several obvious bugs with the implementation (e.g., entering any password from any account appears to decrypt the results of the last computation conducted). Because of this we will focus on the planned version of SCRAM, discussed in 3.2. Additionally, our discussion of security policy will not be focused on implementation issues of MPC via TFHE, issues with backend C++ (e.g., buffer overflow attacks), issues with browser security (e.g., CSRF or phishing), or database attacks. These are engineering challenges present in any product built using these tools and the existence of these issues represent implementation bugs rather than design choices.

3.2 Planned Version of SCRAM

In the planned version of SCRAM there are several clients and one central SCRAM server which clients would connect to. Once a client connects, they can begin the key generation process, generating a public and private key share and sending their generated public key share to the server. As soon as the server receives all of the public key shares, it uses them to generate a shared public key which will be known to all clients; the corresponding shared private key is not known to any one party but split between the clients. Each client then encrypts their data using this shared public key and sends their encrypted data to the server. The server then applies the desired operation on all the encrypted data and sends the resulting ciphertext to all clients. Each client uses their own private key share to partially decrypt the ciphertext, resulting in a decryption share. The clients then publish their decryption shares to all other clients, facilitated by the server; each client can then combine all the decryption shares to obtain the plaintext result of the operation, as detailed in 2.3.4.

Each message should include some authentication, such as a signature, to validate one is receiving information from the correct party in order to prevent MITM attacks. This is discussed further in 3.3.1.

3.2.1 Versions

SCRAM has several versions of planned development. Each version details where the secret key shares are computed and stored. Currently SCRAM is on version 1, intended as a demo version.

- **Version 1:** Private keys are computed and stored unencrypted in the server's database. This is clearly not very secure as anyone with access to the server can collect all of the participants secret keys (and therefore the shared secret key) and see all sent encrypted data.
- **Version 2:** Private keys are computed and stored in the server's database but now are encrypted with a client-chosen password. Immediately after the key computation finishes, the key is not encrypted yet and someone with server access can steal the secret key. Once the secret key is encrypted, if the client is offline, accessing the secret key is as secure as the password encryption method used. However, when the private key is being accessed, it will be present in the server's memory. [However, when the client sends the password, there is a vulnerability that the password they used in stored locally in the browser's cache.]
- **Version 3:** The private key is still computed on the server but is now stored client-side, completely off the server database. There are still potential issues of a server stealing the secret key before it is sent off to the client or the server secretly storing the key. However, given a trusted server, an attacker must attack each client individually to get access to anyone's secret keys.
- **Version 4:** The private key generation and all client computations are separated from server operations, so that private keying material can remain only with the clients on a client-side server.

3.2.2 Use Cases

Use cases for SCRAM appear wherever there are applications of secure multi-party computation, but with the benefit of requiring relatively few computational resources from the parties themselves due to the central-server model of SCRAM compared to classical MPC constructions. This is particularly relevant for corporate participants in secure environments for whom it may be a challenge to install or maintain new software on internal machines.

Examples may include hospitals or other parties who desire to perform computations on sensitive medical data, businesses who would like to collaborate to share insights or aggregate data without exposing their own internal trade secrets, or performing some kind of industry auditing without revealing information between rival companies. In these cases, the SCRAM provider (e.g. MIT) could provide each party with a low-power commodity machine which could remain completely airgapped from the party's internal systems; the relevant data could then be entered into the machine and it could act as the client in the MPC computation.

3.3 Security Policy

SCRAM's goal is to allow secure computation for *honest but curious* parties such that no client information is revealed to the server and only the end result of the computation is revealed to clients.

As mentioned in 2.5, there are potential privacy issues inherent to MPC related to differential privacy concerns; this section will focus on security characteristics specific to SCRAM.

3.3.1 MITM

We consider man-in-the-middle (MITM) attacks under the assumption that communications between parties are not additionally encrypted or authenticated e.g. via TLS.

In the case that a malicious adversary is able to listen in this way to the communications between all the parties but has no knowledge of any of the clients' secret keys, this malicious party should still not be able to gain access to any of the clients' data. However, if this party is able to listen to all communications between the clients at the final decryption stage when they share their decryption shares, it would be possible for them to obtain all the decryption shares and therefore gain knowledge of the final computation result. This could in fact be carried out while only listening to communications to and from a single client, since this would give them access to all decryption shares; if TLS were being used, this would therefore only require one TLS secret key to be compromised.

We note that if this adversary were able to impersonate a client or the server and send messages, they would be able to act to the other clients as a malicious client or malicious server, allowing attacks as described in the following sections.

3.3.2 Malicious Server

We now consider what can happen in the case of a server which cannot be fully trusted, for example, one which cannot be trusted to perform correct computations, or one which is fully malicious. Note that many of the potential attacks that follow can be mitigated by reducing some degree of centralization or trust in the server.

In the case that the server is fully malicious e.g. under the control of an adversary, there are several potential security issues that could arise. Firstly, it would be possible for the server to connect a client to some fake malicious clients also controlled by the adversary, which could allow the server to gain knowledge of the client's data through the computation (as in 2.5). It would also potentially be possible for the server to send clients a fake public shared key to which it knows the secret key, which would allow it to directly steal client data.

Such attacks could be difficult to detect for an unsuspecting client without additional checks in place. For example, in the case of the first attack, the malicious server could steal one client's data (call them client A) using a fake public key but complete the rest of the computation normally with the real shared key, then send the completed ciphertext back to all clients; all clients would receive the correct result as expected. In fact, if the server sent the same fake shared public key to all clients, it would be able to steal all the client data. This could be prevented if clients published their public keys to each other and computed the shared public key themselves.

In the case of the second attack, it would be possible for clients to detect the breach if they compared their final results, or if the targeted client was able to notice that the data they received from the fake clients was clearly bogus.

Note that both these attacks could be mitigated if the clients participating in the computation knew the identities or public keys of the other participating clients beforehand and could therefore verify their identities.

As noted above, some of these problems can be mitigated by decreasing the amount of centralization in the system; on the extreme end, allowing the parties to each do the computations on their own and having the server simply act as a certificate authority would prevent the server from botching computations e.g. via a software bug, but would lose out on most of the benefits of having the central server in the first place.

Another approach would be to have server-side computation be conducted in a secure hardware enclave so that operations could be trusted and data would not be leaked.

3.3.3 Malicious Client

In the case that an attacker were able to pose as a client, for example, by intercepting and replacing messages between the client and the other parties/server, the attacker would be able to once again gain the result of the final computation. Additionally, if they so desired, the attacker would be able to input bogus data and change the output of the computation in a way that could potentially be hard for the real clients to detect. However, assuming no knowledge of secret keys, the attacker would not be able to gain access to any client data.

4 Conclusion

In this paper, we hoped to present some of the theoretical and practical background necessary for understanding the IPRI group's implementation of SCRAM. In addition, we analyzed some potential issues with a server-based MPC protocol and with general use of MPC protocols. Even using strong cryptographic primitives to construct a theoretically secure MPC protocol, vulnerabilities exist both in distribution of the final function evaluation and trusting a server responsible for key distribution. To address these issues, we proposed some potential fixes, such as following differential privacy techniques to have parties add noise to their data before encrypting or requiring clients to directly share their public key shares. However, security in any MPC system will ultimately be a tradeoff with convenience and function accuracy, as using these methods of privacy protection sacrifice function accuracy and some of the convenience of outsourcing work to a common server. Because of this, the right form of MPC for a particular use will likely depend on the necessary complexity, privacy, and accuracy of the inputs and function for that particular application.

References

[1] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 483–501. Springer, 2012.

[2] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 505–524, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. *Advances in Cryptology–CRYPTO 2012*, 2012.

[4] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology*, page 144, 2012.

[5] Ronald Rivest Yael Kalai. Differential privacy: 6.857 lecture 17. February 2020.