Turning HATE Into LOVE: Homomorphic Ad Hoc Threshold Encryption for Scalable MPC

Leonid Reyzin*, Adam Smith**, and Sophia Yakoubov*

Boston University

Abstract. We explore large-scale fault-tolerant multiparty computation on a minimal communication graph. Our goal is to be able to privately aggregate data from thousands of users — for example, in order to obtain usage statistics from users' phones. To reflect typical phone deployments, we limit communication to the star graph (so that all users only talk to a single central server). To provide fault-tolerance, we require the computation to complete even if some users drop out mid-computation, which is inevitable if the computing devices are personally owned smartphones. Variants of this setting have been considered for the problem of secure aggregation by Chan *et al.* (Financial Cryptography 2012) and Bonawitz *et al.* (CCS 2017). We call this setting Large-scale One-server Vanishing-participants Efficient MPC (LOVE MPC).

We show that LOVE MPC requires at least three message flows, and that a three-message protocol requires some setup (such as a PKI). We then build LOVE MPC with optimal round- and communication- complexity (assuming semi-honest participants and a deployed PKI), using homomorphic ad hoc threshold encryption (HATE). We build the first HATE scheme with constant-size ciphertexts (although the public key length is linear in the number of users). Unfortunately, this construction is merely a feasibility result, because it relies on differing-inputs obfuscation.

We also construct more practical three- and five- message LOVE MPC in the PKI model for addition or multiplication. Unlike in the obfuscation-based construction, the per user message length in these protocols is linear in the number of users. However, the five-message protocol still has constant amortized message length, because only the first two messages are long, but they need to be exchanged only once (i.e., are input-independent and reusable) and thus can be viewed as setup.

* Leonid Reyzin and Sophia Yakoubov were supported in part by NSF grant 1422965.

** Adam Smith was supported in part by NSF awards IIS-1447700 and AF-1763786 and a Sloan Foundation Research Award.

Table of Contents

1 Introduction

Consider a service that has an app with a large smartphone user base. Suppose the service wants to collect aggregate usage statistics, but (for regulatory compliance, or for good publicity, or for fear of becoming a target for attackers and investigators) does not wish to learn the data of any individual user.

Let f be the function whose inputs are individual user data from up to n users and whose output is the aggregated information that the service wants to compute. Naturally, a secure multiparty computation protocol (MPC) for f can be used to provide the desired aggregate output to the service without revealing the inputs of any individual user.[1] However, in this setting, we cannot expect every phone to remain engaged for the duration of the protocol, as phones may go out of signal range or run out of charge. Thus, the protocol must be fault-tolerant: it must go on to completion even if some participants drop out. Moreover, given the large number of parties and the limitations on their computational power, the protocol needs to be efficient for every participating user. The users are assumed to be able to communicate directly only with the service provider.

On the other hand, this setting has its own advantages. The service collecting the data is already powerful enough to connect to and perform work for every user, and thus can be assumed to be have a server (or server farm) that will perform a considerable amount of work in the protocol. Moreover, the users already trust the service to provide the code of the app and thus the implementation of the MPC code. Thus, an assumption that the server is semi-honest (aka honest-but-curious) is reasonable: the service itself does not want to have individual user data, for reasons outlined above, and the service itself is interested in arriving at the correct output. In other words, the service is honest, but does not want to know sensitive data, and thus we need to design protection against honest-but-curious servers. We will also assume that the users are honest-but-curious, as they run the app provided by the service. We call this setting Large-scale One-server Vanishing-participants Efficient MPC (LOVE MPC for short).

Of particular interest in this setting is the problem of computing the sum of the users' inputs for so-called *secure aggregation*. The problem of secure aggregation was first studied by Rastogi and Nath [RN10] and Shi *et al.* [SCR$^+$11]. Chan *et al.* [CSS12] added fault-tolerance to the setting. Elahi *et al.* [EDG14] considered the problem of secure aggregation in the context of anonymous routing. Bonawitz *et al.* [BIK$^+$17] considered the same model as we do here (without formalizing it) with the goal of achieving privacy-preserving federated learning. In this context, it is often the case that the users' inputs come from a constant-size space (e.g., binary), and thus the total sum is at most linear in the number of users.

[1] The question of what can be inferred about the individual users from the output of f is important, but orthogonal to our problem; this question is addressed at the point of choosing which f to compute—for example, by ensuring it is differentially private.

1.1 Our Contributions

In this paper, we formalize LOVE MPC and explore its limitations and possibilities. In Section 4.1, we show that three message flows are necessary, and that if LOVE MPC uses only three message flows, some setup (e.g. a PKI) is necessary.

We demonstrate two types of three-message (and therefore round-optimal) semi-honest LOVE MPC protocols for addition in the PKI model. The first type is simple and efficient, but requires messages whose size is linear in the number of users. The second type has constant-size messages but is not useable in practice because of heavy-weight tools (namely, differing-inputs obfuscation).

We also demonstrate a simple and efficient five-message semi-honest LOVE MPC protocol for addition over small message spaces (or multiplication) in the PKI model. This protocol consists of two phases: a two-message setup phase, and a three-message computation phase. The setup phase need only be performed once, after which the computation can be repeated many times. It requires linear-size messages only during the setup phase; after that, each computation uses only constant-size messages.

Our PKI model assumes each user has a public-private key pair, and users know the public keys of all the participants in the protocol. Note that we do not assume any correlated randomness: all the keys are generated separately and independently. How public keys are distributed is not important for our purposes; they can be assumed to be available from the semi-honest server, for example. This setup requires no additional trust assumptions and can be viewed simply as one initial communication round-trip that is reusable.

We now describe our technical approach in a bit more detail.

Three-Message LOVE MPC from HATE. To construct our three-message protocols for LOVE MPC, we rely on the following approach. Each user encrypts her input using the public keys of other users, in such a way that any subset of size $t+1$ users can decrypt it, but any smaller subset cannot. The users send their ciphertexts to the server, who homomorphically combines them, in order to get a ciphertext corresponding to the output of f applied to the plaintexts. The server then sends the combined ciphertext to the users, who each decrypt to obtain shares of the output and send them back to the server; the server combines any $t+1$ of these shares to obtain the output.

Thus, the primitive we require is homomorphic ad hoc threshold encryption (HATE for short): homomorphic so the server can compute f without decrypting), ad hoc so the users can have uncorrelated keys, and threshold so $t+1$ users are necessary and sufficient to decrypt. In Section 3.2, we use differing-inputs obfuscation to construct the very first HATE with constant-size ciphertexts. However, since differing-inputs obfuscation is not useable in practice, in Section 3.1 we also build HATE schemes with ciphertexts linear in the number of users, from practical primitives like secret sharing and public key encryption.

Five-Message LOVE MPC from HTE. If we are willing to stray from round optimality, then we can use a threshold encryption scheme that is homomorphic but

not ad hoc by using an additional round-trip to set up correlated randomness. In particular, we use the ElGamal threshold encryption scheme described in Appendix A. Correlated randomness for this scheme can be set up using Shamir secret sharing in two message flows and $\Theta(n)$ communication per user. After these two rounds, the parties can use threshold ElGamal to compute as many multiplications (or additions over small message spaces) as they choose, at the cost of just three rounds and $\Theta(1)$ communication per user. So, the first computation requires five rounds and $\Theta(n)$ communication, but the amortized cost of a computation is just three rounds and $\Theta(1)$ communication per user.

1.2 Related Work

Work Related to LOVE MPC. Bonawitz *et al.* [BIK+17] present a LOVE MPC protocol for vector addition. Their honest-but-curious protocol can be viewed in the PKI model, similar to ours; it requires five messages and linear per-user communication complexity in the PKI model. In contrast, we present simpler protocols that require only three messages (Construction 3), and, at the cost of an additional two-message setup, can have constant per-user communication (Construction 4). Our computational requirements on the users are also lighter in Construction 4, as we require amortized constant computation, while the protocol of Bonawitz *et al.* requires quadratic computation [BIK+17, Figure 3].

Many works have considered variants of our problem. For example, Shi *et al.* [SCR+11] present protocols in a similar client-server model that that are not fault-tolerant. Chan *et al.* [CSS12] present protocols in the same model that are fault-tolerant, but satisfy a different notion of privacy than MPC and require correlated setup. Tolerating vanishing participants in general MPC is considered by Badrinarayanan *et al.* [BJMS18] (who use the term "lazy" instead of "vanishing").

A number of papers propose the use of techniques similar to ours. In particular, the use of multi-key fully homomorphic encryption (FHE) for round-efficient multi-party computation has been explored by Mukherjee and Wichs [MW16]; the use of threshold FHE was considered by Boneh *et al.* [BGG+18]; and the combination of threshold and multi-key properties for FHE was considered in Badrinarayanan *et al.* [BJMS18]. None of these works consider the client-server communication model we consider. We use threshold homomorphic (but not fully homomorphic) encryption in all of our LOVE MPC constructions.

Work Related to HATE. Fully homomorphic ATE was demonstrated by Badrinarayanan *et al.* [BJMS18], but with polynomial-size ciphertexts. The share-and-encrypt approach that we use in Construction 1 has also appeared in the past (but without homomorphism)—e.g., in the work of Daza *et al.* [DHMR07].

A number of papers [BZ14,ABG+13,Zha14] use obfuscation to achieve constant-size ciphertexts in broadcast encryption; we use it in Constructions 2 and 5 to achieve constant-size ciphertexts in HATE.

2 Threshold Encryption (TE) Definitions

A threshold encryption scheme [DHMR07] is an encryption scheme where a message is encrypted to a group $\mathcal{R}$ of recipients, and decryption must be done collaboratively by at least $t+1$ members of that group. (This can be defined more broadly for general access structures, but we limit ourselves to the threshold access structure.) We show one example of a threshold encryption scheme (a threshold variant of ElGamal, due to Desmedt and Frankel [DF90]) in Appendix A.

2.1 Threshold Encryption Algorithms

A threshold encryption scheme consists of five algorithms, described below. This description is loosely based on the work of Daza *et al.* [DHMR07], but we modify the input and output parameters to focus on those we require in our primary constructions (Section 3.1), with some additional parameters discussed in the text.

$\mathsf{Setup}(1^k, t) \to (\mathsf{params}, \mathsf{msk})$ is a randomized algorithm that takes in a security parameter k and sets up the global public parameters params for the system, as well as the master secret key msk for key generation.
If $\mathsf{msk} = \perp$, the scheme is *ad hoc*, meaning that there is no master secret key and that each party can set up their own public-private key pair.
For simplicity, we provide Setup with t, and assume that t is encoded in params from hereon out. However, in t-flexible schemes, t may be decided by each sender at encryption time, and should then be an input to Enc (and encoded in the resulting ciphertext). In keyed-sender schemes (where the sender must use their secret key to encrypt and recipients must use the sender's public key to decrypt), t may also be specified in the sender's public key.

$\mathsf{KeyGen}(\mathsf{params}, \mathsf{msk}) \to (pk, sk)$ is a randomized key generation algorithm that takes in the global public parameters params and the master secret key msk and returns a public-private key pair.
If the scheme is ad hoc, KeyGen does not require the master secret key msk. Omitting msk from ad hoc schemes enables individual parties to run KeyGen themselves.
Some schemes require the sender's public key for decryption; we call such schemes *keyed-sender*. If the scheme is keyed-sender, the public and secret keys may each have two parts. Informally, those are the parts of the public key necessary for encryption to that party (and the parts of the secret key necessary for decryption by that party), and the parts of the public key necessary for the decryption of a message from that party (and the parts of the secret key necessary for encryption by that party). We discuss keyed-sender schemes further in Section 3.2.

$\mathsf{Enc}(\mathsf{params}, \{pk_i\}_{i \in \mathcal{R}, |\mathcal{R}| > t}, m) \to c$ is a randomized encryption algorithm that encrypts a message m to a set of public keys belonging to the parties in

the intended recipient set $\mathcal{R}$ in such a way that any size-$(t+1)$ subset of the recipient set should jointly be able to decrypt. We assume t is specified within params, but it may also be specified within the sender's public key, or (if the scheme is t-flexible) on the fly as an input to Enc itself.

In *keyed-sender* schemes, Enc may also require the sender's private key sk_{Sndr}.

$\mathsf{PartDec}(\mathsf{params}, \{pk_i\}_{i\in\mathcal{R}}, sk_j, c) \to d_j$ is an algorithm that uses a secret key sk_j belonging to one of the intended recipients to get a partial decryption d_j of the ciphertext c. This partial decryption can then be combined with t other partial decryptions to recover the message.

In *keyed-sender* schemes, $\mathsf{PartDec}$ may also require the sender's public key pk_{Sndr}.

$\mathsf{FinalDec}(\mathsf{params}, \{pk_i\}_{i\in\mathcal{R}}, c, \{d_i\}_{i\in\mathcal{R}'\subseteq\mathcal{R},|\mathcal{R}'|>t}) \to m$ is an algorithm that combines $t+1$ or more partial decryptions to recover the message m.

Not all threshold encryption schemes allow/require all of the algorithm inputs. Sometimes disallowing an input can make the scheme less flexible, but, on the other hand, sometimes schemes that do not rely on certain inputs have an advantage.

More Flexibility: Unneeded Inputs. Most threshold encryption schemes in the literature require a trusted central authority who holds the master secret key msk to be the one to run the key generation algorithm for every party. This is often not ideal; in many scenarios, such a trusted central authority does not exist. We call a threshold encryption scheme *ad hoc* if a public-private key pair can be generated without knowledge of a master secret key; that is, if each party is able to generate their keys independently. We use the acronym ATE to refer to an ad hoc threshold encryption scheme.

Secondly, requiring both decryption algorithms ($\mathsf{PartDec}$ and $\mathsf{FinalDec}$) to be aware of the set of public keys belonging to individuals in the set $\mathcal{R}$ of recipients can be limiting. We call a threshold encryption scheme $\mathcal{R}$-*oblivious* if neither partial decryption nor final decryption uses this information. On the surface, it looks like an $\mathcal{R}$-oblivious scheme should require less communication, since the sender would never need to communicate $\mathcal{R}$ to the recipients. However, in Appendix B we show a lower bound on the ciphertext size in an $\mathcal{R}$-oblivious scheme that is linear in the size of the recipient set.

Less Flexibility: Disallowed Inputs. In the algorithms described above, encryption takes in a set of public keys, giving the sender control over the recipient set $\mathcal{R}$. We call a scheme in which the sender can choose the recipient set $\mathcal{R}$ as a subset of the universe of individuals at encryption time (but after key generation) an $\mathcal{R}$-*flexible* scheme. A scheme that is not $\mathcal{R}$-flexible would simply have each sender encrypt to the entire universe of individuals every time. Additionally, we call a threshold encryption scheme in which the sender can choose t at encryption time a t-*flexible* scheme. A scheme that is not t-flexible would have a fixed t, encoded either in the public parameters or in senders' public keys.

Note that any secure ad hoc threshold encryption scheme is $\mathcal{R}$-flexible, since otherwise, an adversary would be able to decrypt any ciphertext simply by generating enough key pairs. However, not all $\mathcal{R}$-flexible schemes are ad hoc; in fact, most broadcast encryption schemes (which are threshold encryption schemes with $t = 0$) are $\mathcal{R}$-flexible but not ad hoc. In this paper, we focus on threshold encryption schemes that are ad hoc (and thus $\mathcal{R}$-flexible), but not $\mathcal{R}$-oblivious.

2.2 Homomorphic Threshold Encryption

In order to use ad hoc threshold encryption for multi-party computation, we need it to be *homomorphic*. There are three natural notions of homomorphism:

1. Homomorphism over ciphertexts, which is the notion typically considered;
2. Partial decryption homomorphism, which we introduce in this paper; and
3. Server-aided homomorphism, which we also introduce in this paper.

We use the acronym HATE to refer to an ad hoc threshold encryption scheme that has any of these notions of homomorphism. We informally describe all three notions below.

Definition 1. *An* $\mathcal{F}$-homomorphic *threshold encryption scheme additionally has the following algorithm:*

$\mathsf{Eval}(\mathsf{params}, \{pk_i\}_{i\in\mathcal{R}}, [c_1, \ldots, c_l], f) \to c^*$ *is an algorithm that, given* l *ciphertexts and a function* $f \in \mathcal{F}$*, computes a new ciphertext* c^* *which decrypts to* $f(m_1, \ldots, m_l)$ *where each* c_q*,* $q \in [1, \ldots, l]$ *decrypts to* m_q*.*

Definition 2. *A* $\mathcal{F}$-partial decryption homomorphic *threshold encryption scheme additionally has the following algorithm:*

$\mathsf{PdecEval}(\mathsf{params}, \{pk_i\}_{i\in\mathcal{R}}, sk_i, [d_{i,1}, \ldots, d_{i,l}], f) \to d_i^*$ *is an algorithm that, given* l *partial decryptions and a function* $f \in \mathcal{F}$*, computes a new partial decryption* d_i^* *which can be used together with other partial decryptions to recover* $f(m_1, \ldots, m_l)$*, where each* $d_{i,q}$*,* $q \in [1, \ldots, l]$ *is party* P_i*'s partial decryption of an encryption of* m_q*.*

Both the ciphertext c^* produced by Eval and the partial decryption d_i^* produced by $\mathsf{PdecEval}$ should be small — that is, they should have size polynomial in $|\mathcal{R}|$ and k but independent of f and l. Notice that this does not preclude ciphertext growth; for instance, in a homomorphic scheme, a fresh ciphertext might have size independent of $|\mathcal{R}|$, and the output of Eval might have size linear in $|\mathcal{R}|$. We draw a line by calling objects that have size polynomial in k but independent of all other parameters *compact*, and objects that have size polynomial in both k and $|\mathcal{R}|$ *semi-compact*. The outputs of Eval and $\mathsf{PdecEval}$ need only be semi-compact.

The third notion of homomorphism is *server-aided homomorphism*, which is homomorphism with an additional efficiency requirement. If a threshold encryption scheme is *$\mathcal{F}$-server-aided homomorphic*, then the output c^* of Eval (which

itself may only be semi-compact) can be split into compact components $\{c_i^*\}_{i\in\mathcal{R}}$ such that every recipient P_i, $i \in \mathcal{R}$ should then be able to run $\mathsf{PartDec}$ given just one compact component c_i^*. Any homomorphic scheme that operates on compact ciphertexts and produces another compact ciphertext is also server-aided homomorphic; however, a homomorphic scheme that produces semi-compact ciphertexts may also be server-aided homomorphic, as long as the output ciphertexts can be split up into compact components.

The motivation for this notion of homomorphism is that typically, it is desirable for any ciphertexts that are sent between parties to be as short as possible (preferably compact) in order to save on communication complexity. Semi-compact ciphertexts that need to be sent to multiple recipients can be expensive; however, even if the ciphertext is semi-compact, if each recipient only needs one compact component then in terms of communication complexity this can be as good as having compact ciphertexts. We describe a server-aided homomorphic ad hoc threshold encryption scheme in Section 3.2.2.

2.3 Threshold Encryption Security

We use the *semantic security* definition of Boneh *et al.* [BGG+18] for threshold encryption schemes.[2]

Definition 3 (Static Semantic Security). *A threshold encryption scheme* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{PartDec}, \mathsf{FinalDec})$ *is* (n, t)-statically semantically secure *if for all sufficiently large security parameters* k*, no efficient adversary* $\mathcal{A}$ *can win the static semantic security game described in Figure 1 (with polynomial-size* $\mathcal{U}$ *and* $|\mathcal{R}| = n$*) with probability non-negligibly greater than* $\frac{1}{2}$*.*

Note that this definition of security implies that even when the scheme is ad hoc (and therefore KeyGen can be run by participants independently instead of by a trusted central party), KeyGen is assumed to be run honestly; in particular, public keys cannot be generated based on the knowledge of other public keys. We leave the design of definitions and protocols such that public keys *can* be generated maliciously for future work.

In order to make Definition 3 more analogous to real world situations, it would make sense to additionally allow the adversary to query the challenger on messages of its choice, and receive encryptions of those messages along with all corresponding partial decryptions. For the sake of simplicity, instead of modifying the static semantic security game in Figure 1, we add a second notion that we call *partial decryption simulatability* which implies that having the ability to make such queries will give the adversary no additional information. If a threshold encryption scheme is partial decryption simulatable, then it is possible

[2] This is analogous to the static security definition of Gentry and Waters [GW09] for broadcast encryption. In fact, we borrow the adjective "static" from their definition. We can also define adaptive semantic security by allowing the adversary to provide the set of corrupt parties $\mathcal{C}$ after seeing the set of all public keys; however, in this paper we used static, not adaptive security.

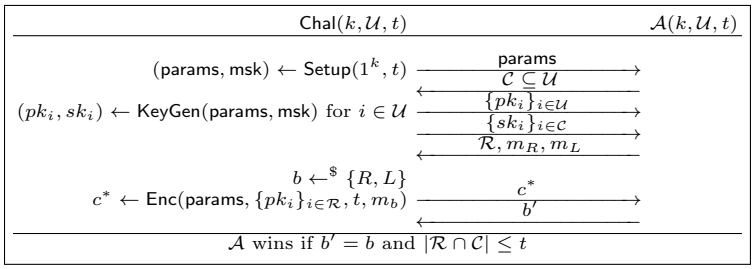

Fig. 1: Static Semantic Security Game for Threshold Encryption

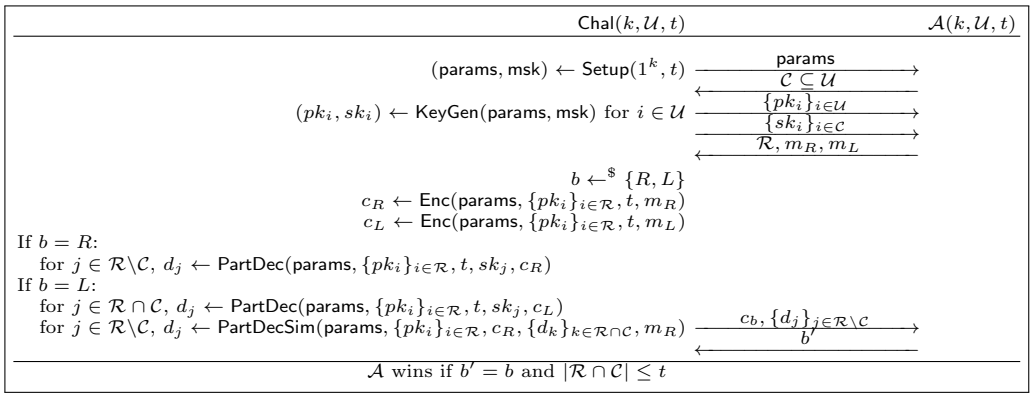

Fig. 2: Static Partial Decryption Simulatability Game for Threshold Encryption

to simulate remaining partial decryptions given t or fewer partial decryptions, a ciphertext, and a desired plaintext output. Our partial decryption simulatability is similar to, but stronger than, simulatability of partial decryption defined in [MW16], where only a single partial decryption can be simulated.

Definition 4. *A threshold encryption scheme* (Setup, KeyGen, Enc, PartDec, FinalDec) *is* (n, t)-statically partial decryption simulatable *if there exists an efficient algorithm* PartDecSim *such that for all sufficiently large security parameters k, no efficient adversary $\mathcal{A}$ can win the game described in Figure 2 (with polynomial-size $\mathcal{U}$ and $|\mathcal{R}| = n$) with probability non-negligibly greater than $\frac{1}{2}$.*

Putting it all together, we say that a threshold encryption scheme is has static security if it meets both of the above definitions.

Definition 5. *A threshold encryption scheme* (Setup, KeyGen, Enc, PartDec, FinalDec) *is* (n, t)-statically secure *if it is both* (n, t)-statically semantically secure *(Definition 3) and* (n, t)-partial decryption simulatable *(Definition 4).*

3 Homomorphic Ad Hoc Threshold Encryption (HATE) Constructions

In this section, we describe some homomorphic ad hoc threshold encryption (HATE) constructions. The table in Figure 3 summarizes their properties. The first row of the table describes prior work, which focuses on *fully* homomorphic ad hoc threshold encryption (FHATE).[3]

In this paper, we consider two categories of additively-homomorphic ATE schemes: those with low *concrete* communication cost, and those with optimal *asymptotic* communication cost. Rows two and three of the table in Figure 3 describe two HATE instantiations — both based on share-and-encrypt (Construction 1) — which, despite their $\Theta(n)$-size ciphertexts, are efficient enough to be used in some scenarios.

The last row of the table ("obfuscation-based HATE", Construction 5) describes the first HATE scheme which has constant-size ciphertexts and partial decryptions. (It is also the first ATE scheme, homorphism or no, with these properties.) Unfortunately, it is a feasibility result more than anything else. It is not useable in practice, since it leverages differing-inputs obfuscation (diO) which currently has no practical instantiations.

Name	pk size	sk size	ctext size	pdec size	homomorphism	message space size	assumption family	t-Flexible ?	$\mathcal{R}$-Oblivious ?
FHATE [BJMS18]	$\Theta(1)$	$\Theta(1)$	$poly(n)$	$poly(n, l)$	any	any	lattices	yes	yes
Shamir-and-ElGamal (Const. 1, Appendix E.1)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	additive	small	DDH	yes	yes
CRT-and-Paillier (Const. 1, Appendix E.2)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	additive	any	factoring	yes	no
obfuscation-based HATE (Const. 5, Section 3.2 and Appendix G)	$poly(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	additive*	small	diO	no	no

Fig. 3: A Summary of Homomorphic Ad Hoc Threshold Encryption Constructions. n refers to the number of parties, and l refers to the number of ciphertexts. (*) For the obfuscation-based scheme, homomorphism can be applied to an expanded ($\Theta(n)$) form of the ciphertext (the scheme is server-aided homomorphic).

3.1 HATE from Homomorphic Encryption and Secret Sharing

One natural way to build ATE is to use a threshold secret sharing scheme SS together with a public-key encryption scheme PKE, as in the work of Daza *et*

[3] There is another paper, due to Boneh *et al.* [BGG+18], that discusses fully homomorphic ad hoc threshold encryption, but because the homomorphism can only be applied to one ciphertext at a time in their scheme, it is not useable to instantiate LOVE MPC and we thus omit it from the table.

$\mathsf{Setup}(1^k)$:
- $\mathsf{params}_{\mathsf{PKE}} \leftarrow \mathsf{PKE.Setup}(1^k)$
- $\mathsf{params}_{\mathsf{SS}} \leftarrow \mathsf{SS.Setup}(1^k)$
- Return $\mathsf{params} = (\mathsf{params}_{\mathsf{PKE}}, \mathsf{params}_{\mathsf{SS}})$

$\mathsf{KeyGen}(\mathsf{params})$:
- Return $(pk, sk) \leftarrow \mathsf{PKE.KeyGen}(\mathsf{params}_{\mathsf{PKE}})$

$\mathsf{Enc}(\mathsf{params}, \{pk_i\}_{i \in \mathcal{R}}, t, m)$:
- $\{[m]_i\}_{i \in \mathcal{R}} \leftarrow \mathsf{SS.Share}(|\mathcal{R}|, t, m)$
- Return $c \leftarrow \{\mathsf{PKE.Enc}(pk_i, [m]_i)\}_{i \in \mathcal{R}}$

$\mathsf{PartDec}(\mathsf{params}, sk_i, c_i)$:
- Return $d_i \leftarrow \mathsf{PKE.Dec}(sk_i, c_i)$

$\mathsf{FinalDec}(\mathsf{params} = 1^k, \{d_i\}_{i \in \mathcal{R}' \subseteq \mathcal{R}, |\mathcal{R}'| > t})$:
- Return $m \leftarrow \mathsf{SS.Reconstruct}(\{d_i\}_{i \in \mathcal{R}' \subseteq \mathcal{R}, |\mathcal{R}'| > t})$

Construction 1: Share-and-Encrypt Ad Hoc Threshold Encryption

al. [DHMR07]. The idea is to secret share the message, and to encrypt each share to a different recipient using their public key; therefore, we call this the *share-and-encrypt* construction. If the secret sharing and encryption schemes are homomorphic in compatible ways, the share-and-encrypt construction is a Homomorphic ATE (HATE).

We assume familiarity with public key encryption and secret sharing. We use one non-standard property of secret sharing, which is *share simulatability*. Informally, a share simulatable t-out-of-n threshold secret sharing scheme ($\mathsf{SS.Share}$, $\mathsf{SS.Reconstruct}$) has a third algorithm $\mathsf{SS.SimShares}$ which takes in a message m and t or fewer honestly generated shares for a different message m', and generates the remaining shares such that all of the shares together are indistinguishable from an honestly generated sharing of m. $\mathsf{SS.SimShares}$ is only used in the proofs, not in the construction. We describe share simulatability in more detail in Appendix C.

Let ($\mathsf{PKE.KeyGen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec}$) be our public-key encryption scheme, and let ($\mathsf{SS.Share}, \mathsf{SS.Reconstruct}$) be our share simulatable threshold secret sharing scheme. We also allow algorithms $\mathsf{PKE.Setup}$ and $\mathsf{SS.Setup}$, which handle global setup for the encryption and secret sharing scheme, respectively. The share-and-encrypt ad hoc threshold encryption scheme is formally defined in Construction 1.

Theorem 1. *Share-and-encrypt (Construction 1) is a (n, t)-statically secure (Definition 5) ATE, as long as SS is a secure share simulatable t-out-of-n secret sharing scheme, and PKE is a CPA-secure public key encryption scheme.*

We prove Theorem 1 in Appendix D. In Appendix E, we describe two homomorphic instantiations of the share-and-encrypt ATE:

1. *Shamir-and-ElGamal* uses exponential Shamir secret sharing and the ElGamal public key encryption scheme, and

2. *CRT-and-Paillier* uses Chinese Remainder Theorem secret sharing and a variant of Paillier encryption.

Theorem 2. *Shamir-and-ElGamal (Appendix E.1) is an additively homomorphic ad hoc threshold encryption scheme for a polynomial-size message space.*

In Shamir-and-ElGamal we are limited to polynomial-size message spaces since final decryption uses brute-force search to find a discrete log. Jumping ahead to LOVE MPC, polynomial-size message spaces are still useful in many applications, as explained in the introduction. Moreover, the server already does work that is polynomial in the number of users, so asking it to perform another polynomial computation is not unreasonable.

Theorem 3. *CRT-and-Paillier (Appendix E.2) is an additively homomorphic ad hoc threshold encryption scheme.*

Both Shamir-and-ElGamal and CRT-and-Paillier are ad hoc threshold encryption schemes by Theorem 1; the homomorphisms in Theorems 2 and 3 follow from the homomorphisms of the underlying encryption and secret sharing schemes.

Communication Complexity. The share-and-encrypt ad hoc threshold encryption scheme has ciphertext size $\Theta(n)$ (asymptotics ignore the security parameter). On the other hand, all private and public keys remain constant-size.

Flexibility. If the secret sharing scheme SS does not require any setup (or requires setup independent of t), the share-and-encrypt scheme is t-flexible, since the sender can decide which threshold t to use at encryption time. Shamir-and-ElGamal is also $\mathcal{R}$-oblivious, since we are able to omit $\{pk_i\}_{i\in\mathcal{R}}$ as an input both to $\mathsf{PartDec}$ and to $\mathsf{FinalDec}$. However, CRT-and-Paillier is not $\mathcal{R}$-oblivious, since parties' moduli, which are part of their public keys, are necessary for $\mathsf{FinalDec}$.

Homomorphism. Depending on the homomorphisms of the underlying secret sharing and encryption schemes, the share-and-encrypt construction can have various homomorphisms. (We discuss specific instantiations in Appendix E.)

Notice that we are able to omit all but the relevant part of the ciphertext as input to $\mathsf{PartDec}$ for each party (where the relevant part is the one encrypted under their key), making the scheme server-aided homomorphic. This further saves on communication in some contexts (Section 4.3).

Finally, since each partial decryption is simply a secret share, the scheme is partial decryption homomorphic in any way that the secret sharing scheme is homomorphic.

3.2 HATE from Differing Inputs Obfuscation

In this section, we introduce the first ad hoc threshold encryption construction with ciphertext size that is independent of the number of parties (at the expense of linear-size public keys). Because our construction is based on differing-inputs obfuscation (diO), its main purpose is to demonstrate that linear-size ciphertexts are not inherent, and a general lower bound is unlikely.

3.2.1 Background Informally, differing-inputs obfuscation [BGI+01,ABG+13] is a way to *obfuscate* a program in such a way that no adversary can tell the difference between the obfuscations of two programs of the same size as long as it is hard to find inputs on which the two programs differ. In other words, if an adversary can distinguish between the obfuscations of programs P_R and P_L ($|P_R| = |P_L|$), then we can use that adversary to recover an input x such that $P_R(x) \neq P_L(x)$.

We also make use of puncturable pseudorandom functions (PPRFs). A PPRF [KPTZ13,BW13,BGI14b,SW14] is a pseudorandom function (PRF) whose keys can be punctured. Let $k\{x\}$ denote the PRF key k punctured at point x. Then $\mathsf{PPRF}_k(x') = \mathsf{PPRF}_{k\{x\}}(x')$ for all $x' \neq x$, but given $k\{x\}$, $\mathsf{PPRF}_k(x)$ is indistinguishable from random.

Finally, we also use accumulators, which provide a compact representation of an arbitrarily large set which allows proofs of membership in the set. One example of an accumulator is a Merkle hash tree [Mer88].[4] The root of the Merkle tree is a short value a which represents the set of leaf elements, and a membership witness for a leaf element is its authenticating path w in the tree.[5] Let $\mathsf{verify}(a, w, x)$ be the algorithm that verifies the membership of element x in the accumulator a using witness w. Reyzin and Yakoubov [RY16] provide more detailed background on accumulators.

We will also use standard semantically-secure public-key encryption [GM84], existentially unforgeable signatures [GMR88], and Shamir secret sharing [Sha79]. We do not provide formal definitions for the primitives we use, as these are standard and available in the referenced literature.

3.2.2 Building ATE from Obfuscation The only asymptotic communication inefficiency in the share-and-encrypt HATE constructions of Section 3.1 comes from the $\Theta(n)$-size ciphertext. We can try to compress the ciphertext using obfuscation; instead of using the encrypted shares as the ciphertext, we can try to use an obfuscated program that *outputs* one encrypted share at a time given an appropriate input (such as receiver secret and public keys, and proof of the receiver's membership in the recipient set $\mathcal{R}$).

However, because it is difficult to obfuscate a secret sharing scheme without having the obfuscated program be of size linear in the threshold (because of the amount of randomness required for sharing), and we want our ciphertexts to be compact, we instead obfuscate a message- and recipient-set- agnostic program.

This obfuscation will be of size linear in the threshold, but it doesn't need to be part of the ciphertext; instead, each party can include such an obfuscated program just once in its public key. One can think of the obfuscated program in

[4] We can also use RSA accumulators [Bd94], putting the modulus in the relevant public key.

[5] For our purposes, is important that this accumulator be *deterministic*. So, if we use Merkle trees as our accumulator, we always order the leaves of our Merkle tree in increasing lexicographic order.

the sender's public key as a "horcrux".[6] The sender stores some of its secrets in this obfuscated program, and when it encrypts a message, the sender includes just enough information in the ciphertext that the obfuscated program can do the rest of the work.

Of course, the obfuscated program should make sure to only produce outputs (a) for intended recipients $i \in \mathcal{R}$, and (b) for ciphertexts actually encrypted by the sender.

To achieve (a) — that is, to make sure that the obfuscated program only produces outputs for intended recipients — the sender will an use accumulator. The obfuscated program requires three input values to ensure that it only produces outputs for the intended recipients:

1. A pair of recipient keys (pk_i, sk_i). The program checks that the public and secret keys match. We assume the existence of an algorithm $\mathsf{matches}$ that performs this check.
2. An accumulator a of recipient public keys pk_j for $j \in \mathcal{R}$.
3. A membership witness w for pk_i in the accumulator a. The program checks that $\mathsf{verfiy}(a, w, pk_i) = 1$.

To achieve (b) — that is, to make sure that the obfuscated program only produces outputs for ciphertexts actually sent by the sender — the sender generates a signing key $\mathsf{SIG}.sk_{\mathsf{Sndr}}$ and a signature verification key $\mathsf{SIG}.pk_{\mathsf{Sndr}}$. It hardcodes $\mathsf{SIG}.pk_{\mathsf{Sndr}}$ in the obfuscated program, and includes a signature as part of the ciphertext. Each ciphertext comprises a sub-ciphertext (encryption of the actual message), an accumulator a, and a signature on both.

Notice that having this obfuscated program as the sender's public key makes the obfuscation-based ATE scheme different from a typical public-key encryption scheme: the ATE scheme is *keyed-sender*, meaning that in order to encrypt a message the sender must use its secret key, and in order to decrypt a message, recipients need to use the sender's public key.

The program each sender must obfuscate and include in their public key is described in Algorithm 1. The obfuscation-based ATE itself is described in Construction 2.

[6] A "horcrux" is a piece of one's soul stored in an external object, according to the fantasy series Harry Potter [Row05].

$\mathsf{KeyGen}(t)$**:**

{The following generates the "receiver" portion of the keys. $\mathsf{PKE}.pk$ is used by others when sending messages to this party, and $\mathsf{PKE}.sk$ is used by this party to decrypt messages from others.}

$(\mathsf{PKE}.pk, \mathsf{PKE}.sk) \leftarrow^{\$} \mathsf{PKE.KeyGen}(1^k)$.

{The following generates the "sender" portion of the keys. $\mathsf{SIG}.sk$ and k' are used by this party when sending messages to others, and $\mathsf{ObfFunc}$ is used by others to decrypt messages from this party.}

$(\mathsf{SIG}.pk, \mathsf{SIG}.sk) \leftarrow^{\$} \mathsf{SIG.KeyGen}(1^k)$.

$k' \leftarrow^{\$} \mathsf{PPRF.KeyGen}(1^k)$.

for $j \in [1, \ldots, t]$ **do**

  $k_j \leftarrow^{\$} \mathsf{PPRF.KeyGen}(1^k)$.

$\mathsf{ObfFunc} \leftarrow \mathsf{diO}(f_{k=(k_1,\ldots,k_t),k',\mathsf{SIG}.pk})$.

return $pk = (\mathsf{PKE}.pk, \mathsf{ObfFunc}), sk = (\mathsf{PKE}.sk, \mathsf{SIG}.sk, k')$.

$\mathsf{Enc}((\mathsf{SIG}.sk_{\mathsf{Sndr}}, k'_{\mathsf{Sndr}}), \{pk_i\}_{i\in\mathcal{R},|\mathcal{R}|\geq t}, m)$**:**

Compute a deterministic accumulator a (e.g. a Merkle hash tree) of $\{(pk_i, i)\}_{i\in\mathcal{R}}$ (where the indices i are, for instance, the indices of the public keys in a lexicographic ordering of all the public keys belonging to recipients in $\mathcal{R}$).

$c_1 \leftarrow^{\$} \{0,1\}^k$.

$c_2 = \mathsf{PPRF}_{k'_{\mathsf{Sndr}}}(c_2) \oplus m$.

$c' = (c_1, c_2)$.

$r \leftarrow^{\$} \{0,1\}^k$.

$\sigma \leftarrow^{\$} \mathsf{SIG.Sign}(\mathsf{SIG}.sk_{\mathsf{Sndr}}, (a, c', r))$.

return $c = (a, c', r, \sigma)$.

$\mathsf{PartDec}(\mathsf{ObfFunc}_{\mathsf{Sndr}}, \{\mathsf{PKE}.pk_j\}_{j\in\mathcal{R}}, \mathsf{PKE}.sk_i, c)$**:**

Parse $(a, c', r, \sigma) = c$.

Recompute a deterministic accumulator a (e.g. a Merkle hash tree) of $\{(\mathsf{PKE}.pk_j, j)\}_{j\in\mathcal{R}}$ (where the indices j are, for instance, the indices of the public keys in a lexicographic ordering of all the public keys belonging to recipients in $\mathcal{R}$). Let w be the witness of $(\mathsf{PKE}.pk_i, i)$ in that accumulator.

$[m]_i \leftarrow \mathsf{ObfFunc}_{\mathsf{Sndr}}(a, c', r, \sigma, \mathsf{PKE}.pk_i, \mathsf{PKE}.sk_i, i, w)$.

$d_i = (i, [m]_i)$.

return d_i.

$\mathsf{FinalDec}(\{d_i\}_{i\in\mathcal{R}'\subset\mathcal{R}})$**:**

Perform Shamir reconstruction to recover the message m.

Construction 2: Obfuscation-Based ATE

Algorithm 1 $f_{k,k',\mathsf{SIG}.pk}(a, c', r, \sigma, \mathsf{PKE}.pk, \mathsf{PKE}.sk, i, w)$

The following values are hardcoded in the program:

- puncturable PPRF keys $k = (k_1, \ldots, k_t)$ {These are used for Shamir sharing (for generating polynomial coefficients).}
- puncturable PPRF key k' {This is used for decrypting the message.}
- signature verification key $\mathsf{SIG}.pk$

The following values are expected as input:

- accumulator a
- ciphertext $c' = (c_1, c_2)$
- random value r
- signature σ
- public encryption key $\mathsf{PKE}.pk$
- secret decryption key $\mathsf{PKE}.sk$
- index i
- witness w

if $(\mathsf{matches}(\mathsf{PKE}.sk, \mathsf{PKE}.pk))$ and $(\mathsf{verify}(a, w, (\mathsf{PKE}.pk, i)))$ and $(\mathsf{SIG.Verify}(\mathsf{SIG}.pk, (a, c', r), \sigma))$ **then**
  $(c_1, c_2) = c'$
  $w = \mathsf{PPRF}_{k'}(c_1)$
  $m = w \oplus c_2$
  for $j \in [1, \ldots, t]$ **do**
    $\mathsf{coef}_j = \mathsf{PPRF}_{k_j}(r)$
  $\mathsf{coef} = (\mathsf{coef}_1, \ldots, \mathsf{coef}_t)$
  return $m + \sum_{j \in [1,\ldots,t]} \mathsf{coef}_j i^j$
  {This returns the ith Shamir share of m, $[m]_i$. Arithmetic is done in a large enough prime-order finite field.}

Notice that Algorithm 1 expects and makes use of an index i for each recipient. This index can be any non-zero element that is deterministically and uniquely (within the recipient set $\mathcal{R}$) associated with the recipient. For instance, it can be the index of $\mathsf{PKE}.pk_i$ in a lexicographic ordering of $\{\mathsf{PKE}.pk_j\}_{j \in \mathcal{R}}$. Once we add server-aided homomorphism and start using Algorithm 4, it becomes important that i come from a small space so as to minimize the size of the ciphertext. So, the above idea of using the index of the recipient's public key in a lexicographic ordering becomes particularly appealing.

Security. We alter the static semantic security game (and partial decryption simulatability game) slightly to accommodate the obfuscation-based ATE construction. Informally, we require the adversary to commit to the recipient set $\mathcal{R}$ earlier (this is necessary for some of the proof hybrids), and we provide the appropriate keys now that the scheme is keyed-sender. We describe the updated games in Appendix F. We call the new notion of security *super-static security*.

Theorem 4. *The obfuscation-based ATE (Construction 2) is* (n, t)*-super-statically secure (Definition 9) for any polynomial* n, t*, as long as* diO *is a secure differing-inputs obfuscation,* PPRF *is a secure puncturable* PRF*,* SIG *is an existentially unforgeable signature scheme,* PKE *is a CPA-secure public key encryption scheme, and the accumulator scheme is secure.*

We prove Theorem 4 in Appendix F.

Communication Complexity. The public keys in the obfuscation-based ATE (Construction 2) are large; because of the obfuscated program, which contains $t + 1$ PPRF keys, the public keys are of size polynomial in t. However, the ciphertexts are constant-size. This is the first ATE with constant-size ciphertexts.

Flexibility. The obfuscation-based ATE is not t-flexible, since the threshold t is fixed within the sender's public key. It is not $\mathcal{R}$-oblivious either, since each receiver has to recompute an accumulator of all receivers' public keys.

Adding Server-Aided Homomorphism. Since partial decryptions are simply Shamir shares of the message, the obfuscation-based ATE is additively partial decryption homomorphic. However, in its current form, it is not homomorphic or server-aided homomorphic. Informally, in order to make it additively homomorphic, we can modify the obfuscated algorithm to:

1. Not require a secret key as input,
2. Return encryptions of the secret shares instead of plaintext secret shares, and
3. Use a homomorphic encryption scheme PKE.

This modification would make the construction additively *server-aided homomorphic*; a server can save the recipients work by evaluating the obfuscated program to extract encryptions of all recipients' partial decryptions, do homomorphic computation on those partial decryptions (since our PKE scheme is homomorphic, and we already have partial decryption homomorphism), and send all parties their final encrypted partial decryption.

More concretely, we can try using ElGamal encryption [ElG84]. Since ElGamal is multiplicatively homomorphic (not additively homomorphic), we use exponential Shamir sharing to make the homomorphisms play nicely together. Once the obfuscated program is evaluated, we are essentially using the Shamir-and-ElGamal HATE (described in detail in Appendix E.1). In particular, this implies that we are limited to polynomial-size message spaces, since final decryption uses brute-force search to find a discrete log (see discussion after Theorem 2).

In Appendix G we give more details about this modification. Construction 5 describes the new additively server-aided homomorphic HATE; Algorithm 4 describes the new program that needs to be obfuscated and included in each sender's public key. Note that this program is now of size linear in n, not t; this means that public keys now have size $poly(n)$.

4 Large-scale One-server Vanishing-participants Efficient MPC (LOVE MPC)

Large-scale One-server Vanishing-participants Efficient MPC (LOVE MPC) is different from more traditional MPC in two ways: (1) in addition to tolerating corruptions, it tolerates some number of parties who vanish (i.e., drop out mid-computation), and (2) only the server learns the output. Our model is influenced by the work of Badrinarayanan *et al.* [BJMS18], which introduces the notion of "lazy parties" who may drop out during the protocol execution.

4.1 Lower Bounds

We show lower bounds both for the number of message flows in a LOVE MPC construction, and for the setup requirements.

Theorem 5. *For many functions (including addition), a LOVE MPC cannot be instantiated in fewer than three message flows, and if only three flows are used, then setup (e.g. correlated randomness or PKI) is unavoidable.*

Proof. We prove this theorem in two parts.

Lower Bounds on Number of Message Flows. A one-message protocol (where each user sends the server a single message, as in non-interactive MPC (NIMPC) [BGI+14a]) is impossible in our setting for many functions f, for the following reason. In a one-message protocol, the users would all send a single message to the server, who would compute the desired output. However, if the protocol is fault-tolerant, the set of participating users cannot be known in advance. Thus, an honest-but-curious server would be able to compute f on many different subsets of participating users, simply by ignoring some of the received messages. For example, if f is simply the sum of the users' individual values, the server could compute f both with and without a particular user present, thus learning every user's input.

A two-message protocol does not make sense, since a second message flow would involve the server sending the users messages. A server-to-user message before the user-to-server message does not solve the above problem, and a server-to-user message after the user-to-server message cannot affect the output, since the server should be the one to arrive at the output. We conclude that a LOVE MPC construction requires at least three message flows.

Lower Bounds on Setup Assumptions. A three-message protocol without any joint setup (e.g. correlated randomness or PKI) allows the server to perform what is essentially a Sybil attack. By fault-tolerance, the output should still be computable if a few participants drop out after sending the first message. Moreover, the output should not change depending on which participants drop out in the third message flow; otherwise, the honest-but-curious server can pretend some users dropped out and see how the output changes (just like in the argument against one-message protocols). Therefore, the output should be fixed as

soon as the second flow messages are sent by the server. (Generalizing to more than three message flows, the output should be fixed as soon as the server sends its last message.) This feature enables an honest-but-curious server to compute f on any single real user's input combined with inputs of the server's choice, as follows. After receiving the first message from a real user, the server will simulate the first message of $n-1$ users with inputs of the server's choice, and then simulate the rest of the messages of the protocol as if the real user dropped out before sending its third message. As long as the protocol can tolerate a single user dropping out, the server will be able to compute the desired output. We conclude that a three-message LOVE MPC construction requires some setup.

4.2 Definitions

Our ideal functionality, described in Figure 4, is a variant of the trusted party functionality of Badrinarayanan *et al.* [BJMS18], modified to support only a single output party (the server Srvr) and to allow functions with more than a single bit of output.

Let $\mathcal{U}$ be the set of all parties, $\mathcal{P}$ be the set of parties who do not drop out by the end of the protocol execution, and $\mathcal{C}$ be the set of corrupt parties. For correctness, we require that $|\mathcal{P}| > t$ for a dropout threshold t. For security, we require that $|\mathcal{C}| \leq t_c$ for a corruption threshold t_c. Note that in all of our constructions, we have $t = t_c$.

A static semi-honest adversary $\mathcal{A}$ specifies the following sets: $\mathcal{C} \subseteq \mathcal{U}$ of corrupt parties such that $|\mathcal{C}| \leq t_c$, $\mathcal{D}_{\text{INPUT}} \subseteq \mathcal{U}$ of parties who drop out before the end of the input phase, and $\mathcal{D}_{\text{OUTPUT}} \subseteq \mathcal{U}$ of parties who drop out after the input phase (where $\mathcal{P} = \mathcal{U} \backslash (\mathcal{D}_{\text{INPUT}} \cup \mathcal{D}_{\text{OUTPUT}})$). Only parties who do not drop out before the end of the input phase (that is, $i \in \mathcal{U} \backslash \mathcal{D}_{\text{INPUT}}$) have their inputs included in the computation. The adversary receives the view of all the parties in $\mathcal{C}$.

Informally, a LOVE MPC protocol has static semi-honest security if for all input vectors $\{x_i\}_{i \in \mathcal{U}}$ and all efficient adversaries $\mathcal{A}$, there exists a simulator $\mathcal{S}$ who interacts with the ideal functionality in Figure 4 and can simulate the view of $\mathcal{A}$. Badrinarayanan *et al.* [BJMS18] also discuss security against malicious parties, which we do not address here.

We present our protocols in the PKI model. Because we consider only honest-but-curious attackers, the PKI model does not require any additional trust: the clients could simply exchange public keys via the server in two additional message flows before the start of the protocol. The importance of the PKI model for our protocols is that this exchange is independent of the inputs and needs to happen only once; after that, the protocols can be run repeatedly with the same public keys.

There are multiple ways to model PKI formally: "global" setup (e.g., Canetti and Rabin [CR03], Canetti *et al.* [CDPW07] and Dodis *et al.* [DKSW09]), which uses key registration that is shared by multiple, possibly different, protocols; or "local" setup (e.g., Barak *et al.* [BCNP04]), in which key registration is per protocol instance. In any of these, since our adversary is semi-honest, the simulator is allowed to know the secret keys of the corrupted parties; in addition, local

Functionality $\mathcal{F}_f$, interacting with server Srvr and parties P_i, $i \in \mathcal{U}$.

INIT: On input (INIT) from the simulator $\mathcal{S}$:
1. Initialize an empty map INPUTS from parties to their inputs.

INPUTABORT: On input (INPUTABORT, $\mathcal{D}_{\text{INPUT}}$) from the simulator $\mathcal{S}$, store $\mathcal{D}_{\text{INPUT}}$.
INPUT: On input (INPUT, x_i) from party P_i: Store INPUTS$[i] = x_i$.
OUTPUTABORT: On input (OUTPUTABORT, $\mathcal{D}_{\text{OUTPUT}}$) from the simulator $\mathcal{S}$, store $\mathcal{D}_{\text{OUTPUT}}$.
OUTPUT: On input (OUTPUT) from the simulator $\mathcal{S}$:
1. Remove i from INPUTS for $i \in \mathcal{D}_{\text{INPUT}}$.
2. If $|\mathcal{U} \backslash (\mathcal{D}_{\text{INPUT}} \cup \mathcal{D}_{\text{OUTPUT}})| > t$: compute $y = f(\text{INPUTS})$.
3. Else: set $y = \perp$.
4. Output y to the server Srvr.

Fig. 4: Ideal Functionality $\mathcal{F}_f$ for LOVE MPC Secure Against Semi-Honest Adversaries.

setup means that the security definition is weaker and the simulator is more powerful, because the simulator can simulate the setup and thus is able to know (or even decide) secret keys for the honest parties. The LOVE MPC protocol that we describe in Section 4.3 can be proven secure with global setup, unless it is instantiated with the keyed-sender obfuscation-based HATE (Section 3.2), in which case it requires local setup (but still can be run multiple times with the same PKI).

4.3 Three-Message LOVE MPC from HATE

Let (HATE.Setup, HATE.KeyGen, HATE.Enc, HATE.PartDec, HATE.FinalDec, HATE.Eval) be a homomorphic ad hoc threshold encryption scheme. Assume the HATE has been set up (and so params is publicly available, and contains t), and that each party has already run KeyGen and that everyone's public keys have been distributed through a public key infrastructure. We describe a three-message HATE-based LOVE MPC in Construction 3. When instantiated with Shamir-and-ElGamal or CRT-and-Paillier, we call it Shamir-and-ElGamal LOVE MPC or CRT-and-Paillier LOVE MPC, respectively. As written, Construction 3 does not use keyed-sender HATE, and so cannot be instantiated with obfuscation-based HATE. However, in Section 4.4 we alter Construction 3 to use obfuscation-based HATE and call the result obfuscation-based LOVE MPC.

Theorem 6. *HATE-based LOVE MPC (Construction 3) in the global-setup PKI model returns the correct output of f if fewer than t parties drop out. It is secure against t static semi-honest corruptions as long as* HATE *is a (n, t)-statically secure (Definition 5) $\mathcal{F}$-homomorphic ATE construction such that $f \in \mathcal{F}$.*

Proof. Correctness is true by the correctness of the underlying HATE.

To prove security, we describe a simulator $\mathcal{S}$ in Figure 5. Since we require global setup, $\mathcal{S}$ does not have access to honest parties' secret keys. However, because of the honest-but-curious assumption, $\mathcal{S}$ does see corrupt parties' secret keys and randomness. $\mathcal{S}$ can simulate by encrypting 0 for each honest party in Flow 1 (without knowing their secret keys), and simulating the partial decryptions for each honest party in Flow 3 (again without knowing their secret keys),

Flow 1: Each party P_i sends a message to the server Srvr

Each party P_i, $i \in \mathcal{U}$ does the following:

1. Computes
$$c_i \leftarrow \mathsf{HATE.Enc}(\mathsf{params}, \{pk_i\}_{i\in\mathcal{U}}, x_i).$$
2. Sends c_i to Srvr.

Flow 2: Server Srvr sends a message to each party P_i

Let $\mathcal{D}_{\text{INPUT}} \subseteq \mathcal{U}$ be the set of parties from whom the server Srvr did not receive a ciphertext. Srvr computes the sum ciphertext

$$c \leftarrow \mathsf{HATE.Eval}(\mathsf{params}, \{pk_i\}_{i\in\mathcal{U}}, \{c_i\}_{i\in\mathcal{U}\setminus\mathcal{D}_{\text{INPUT}}}, f)$$

and sends c to all parties $i \in \mathcal{U}\setminus\mathcal{D}_{\text{INPUT}}$.

Flow 3: Each party P_i sends a message to the server Srvr

Each party P_i, $i \in \mathcal{U}\setminus\mathcal{D}_{\text{INPUT}}$ does the following:

1. Computes
$$d_i \leftarrow \mathsf{HATE.PartDec}(\mathsf{params}, \{pk_j\}_{j\in\mathcal{U}}, sk_i, c).$$
2. Sends d_i to Srvr.

The server Srvr computes the output

Let $\mathcal{D}_{\text{OUTPUT}} \subseteq \mathcal{U}\setminus\mathcal{D}_{\text{INPUT}}$ be the set of parties from whom the server Srvr got a ciphertext c_i, but not a partial decryption d_i. As long as $|\mathcal{P} = \mathcal{U}\setminus(\mathcal{D}_{\text{INPUT}} \cup \mathcal{D}_{\text{OUTPUT}})| > t$, Srvr computes

$$y \leftarrow \mathsf{HATE.FinalDec}(\mathsf{params}, \{pk_i\}_{i\in\mathcal{U}}, c, \{d_i\}_{i\in\mathcal{P}}).$$

Construction 3: LOVE MPC for Function f From HATE in Three Rounds

1. The simulator $\mathcal{S}$ sends (INIT) to the ideal functionality.
2. $\mathcal{S}$ runs the adversary $\mathcal{A}$ to determine $\mathcal{C}$, $\mathcal{D}_{\text{INPUT}}$, $\mathcal{D}_{\text{OUTPUT}}$, and sends those to the ideal functionality.
3. [Flow 1] For each honest party P_i, $i \in \mathcal{U} \backslash \mathcal{C}$, $\mathcal{S}$ encrypts 0 in place of the actual input:

$$c_i \leftarrow \mathsf{HATE.Enc}(\mathsf{params}, \{pk_j\}_{j \in \mathcal{U}}, 0).$$

4. [Flow 2] Whether the server Srvr is honest or semi-honest (that is, whether or not its role is played by the simulator), it will correctly compute the sum ciphertext c and send it to all parties.
5. [Flow 3] For each corrupt party P_i, $i \in \mathcal{C}$, since the simulator $\mathcal{S}$ knows sk_i and the randomness used by the adversary, $\mathcal{S}$ can compute the partial decryption $d_i \leftarrow \mathsf{HATE.PartDec}(\mathsf{params}, \{pk_j\}_{j \in \mathcal{U}}, sk_i, c)$, which is guaranteed to equal the one the corrupt party will send the server Srvr in the final flow. ($\mathcal{S}$ can do this even if the adversary is rushing.) However, since $\mathcal{S}$ does not know the honest parties' keys, $\mathcal{S}$ must simulate their partial decryptions. $\mathcal{S}$ sends (OUTPUT) to the ideal functionality (note that the output is fixed at this point, since it is fixed as soon as the server sent the ciphertext), learns the actual output y, and computes the simulated partial decryptions for the honest parties by using the partial decryption simulatability (Definition 4) of the HATE. That is, the simulator runs

$$\{d_i\}_{i \in \mathcal{U} \backslash \mathcal{C}} \leftarrow \mathsf{HATE.PartDecSim}(\mathsf{params}, \{pk_i\}_{i \in \mathcal{U}}, c, \{d_i\}_{i \in \mathcal{C}}, y).$$

   By partial decryption simulatability, these will be indistinguishable from genuine partial decryptions.
6. [Output Computation] Whether the server Srvr is honest or semi-honest, it will correctly compute the output from the partial decryptions it receives.

Fig. 5: Simulator $\mathcal{S}$ for LOVE MPC from HATE

by first performing partial decryption on behalf of the corrupt parties using their secret keys and randomness.

Notice that the only points in which the simulation differs from a real execution view is Flow 1, when the simulator encrypts 0s instead of the actual inputs, and Flow 3, when the simulator simulates partial decryptions instead of using genuine ones. The simulated corrupt parties' view is indistinguishable from a real view by CPA security and partial decryption simulatability, respectively.

Efficiency. Shamir-and-ElGamal LOVE MPC and CRT-and-Paillier LOVE MPC require $\Theta(n)$ communication per party, where $n = |\mathcal{U}|$. Since ciphertexts are $\Theta(n)$ in size, each party sends a $\Theta(n)$-size message in Flow 1, and receives a $\Theta(n)$-size message in Flow 2. However, we can leverage the server-aided homomorphism of share-and-encrypt and save some concrete cost by having the server only send each party the relevant part of the ciphertext in Flow 2; that is, the encryption of their secret share.

4.4 Three-Message LOVE MPC from Keyed-Sender Server-Aided Homomorphic ATE

The LOVE MPC protocol above (Construction 3) uses HATE that is *not keyed-sender* (that is, the sender does not need to use their own secret key to encrypt); so, it fits perfectly with our share-and-encrypt HATE (Construction 1), but not with our obfuscation-based HATE (Construction 5). We can modify it to use

a HATE that is *keyed-sender* by adding sk_i as an input to HATE.Enc. When instantiated with obfuscation-based HATE, we call it obfuscation-based LOVE MPC. Obfuscation-based LOVE MPC will still be secure, with the caveat that now, the simulator will need access to all secret keys, because otherwise it will not be able to simulate honest parties' encryptions. This means that obfuscation-based LOVE MPC requires local setup.

Note that obfuscation-based HATE has super-static security (Definition 9) instead of static security; this means that the adversary must commit to the recipient set before seeing public keys. In particular, when we build LOVE MPC out of this HATE construction, a given setup instance can only be used for LOVE MPC among a fixed set of recipients.

Efficiency. Obfuscation-based LOVE MPC requires only constant communication per party.

4.5 Five-Message LOVE MPC from Homomorphic Threshold Encryption

Given a threshold encryption scheme that is homomorphic but lacks the ad hoc property, we can achieve a LOVE MPC five-message protocol for multiplication in two phases. In the first phase, the parties establish some correlated randomness (which can be reused). In the second phase, the parties leverage the correlated randomness and use (non ad hoc) multiplicatively-homomorphic threshold encryption to compute on their inputs in three message flows. Note that the first phase is reusable; the second phase can be re-executed multiple times.

We can use the multiplicatively-homomorphic ElGamal-based threshold encryption construction TEG due to Desmedt and Frankel [DF90], described in Appendix A. The scheme operates over a $\mathcal{G}$ of prime order p with generator g in which the decisional Diffie-Hellman problem is assumed to be hard. We assume that $\mathcal{G}$, p, and g are known to everyone; we also assume that each party P_i already has a key pair (pk_i, sk_i) for some semantically secure encryption scheme, and that pk_i is known to everyone.

We show this protocol, which we call Reusable Threshold ElGamal LOVE MPC, in Construction 4. In the first phase the parties run TEG.Setup as well as TEG.KeyGen in two message flows. In the second phase, all parties use the resulting instance of threshold ElGamal to encrypt their values to the joint public key pk, the server homomorphically multiplies them, broadcasts the resulting short ciphertext, and decrypts using the short partial decryptions it gets back.

In Construction 4 we show how the parties can compute multiplication. If the parties want to compute addition over small message spaces instead of multiplication, each party should encrypt g^{x_i} instead of x_i, and after TEG decryption the server can recover the output through brute-force search (same as in Theorem 2).

Theorem 7. *Reusable Threshold ElGamal LOVE MPC (Construction 4) in the global-setup PKI model returns the product of the parties' inputs in $\mathcal{G}$ if fewer*

Flow 1.1 Each party P_i, $i \in \mathcal{U}$ does the following:
1. Picks a random $r_i \leftarrow^{\$} [p]$.
2. Shamir-shares r_i by picking a random polynomial f_i of degree t over the field $\mathbb{Z}_p$ with r_i as its y-intercept, and computing $[r_i]_j = f_i(j)$ for $j \in \mathcal{U}$.
3. Computes $c_{i,j} = \mathsf{PKE.Enc}(\mathsf{PKE}.pk_j, [r_i]_j)$ for $j \in \mathcal{U}$.
4. Sends $(g^{r_i}, \{c_{i,j}\}_{j \in \mathcal{U}})$ to the server Srvr.

Flow 1.2 The server Srvr does the following:
1. Computes the shared public key $pk = \prod_{i \in \mathcal{U}} g^{r_i}$
2. For each $i \in \mathcal{U}$, forwards the ciphertexts $\{c_{j,i}\}_{i \in \mathcal{U}}$ (as well as the shared public key pk) to P_i

Phase 1 Post-Processing Each party P_i, $i \in \mathcal{U}$ stores the shared public key pk, decrypts $c_{j,i}$ to obtain $[r_j]_i$ for all $j \in \mathcal{U}$, and computes its secret key share as $[sk]_i = \sum_{j \in \mathcal{U}} [r_j]_i$.

Flow 2.1 Each party P_i, $i \in \mathcal{U}$ sends $c_i = \mathsf{TEG.Enc}(pk, x_i)$ to Srvr.

Flow 2.2 The server Srvr computes the product ciphertext $c = \mathsf{TEG.Eval}(pk, \{c_i\}_{i \in \mathcal{U}}, \times)$, and sends c to all parties.

Flow 2.3 The parties compute their partial decryptions as $d_i = \mathsf{TEG.PartDec}([sk]_i, c)$ and send d_i to the server Srvr.

Phase 2 Post-Processing Srvr combines the partial decryptions as $y = \mathsf{TEG.FinalDec}(\{pk'_i\}_{i \in \mathcal{R}}, \{d_i\}_{i \in \mathcal{R}' \subseteq \mathcal{R}}, c)$ to obtain the output y.

Construction 4: Reusable Threshold ElGamal LOVE MPC

than t parties drop out. It is secure against t static semi-honest corruptions under the decisional Diffie-Hellman assumption, as long as PKE *is CPA-secure.*

Proof. Correctness is true by the correctness of the underlying primitives.

We informally prove security by describing a simulator $\mathcal{S}$. $\mathcal{S}$ has access to all parties' threshold ElGamal keys $[sk]_i$: because we have a semi-honest adversary, the simulator $\mathcal{S}$ can see all of the corrupt parties' randomness and computation, and because the simulator plays the honest parties' roles in the first phase, $\mathcal{S}$ learns the honest parties' keys as well. $\mathcal{S}$ behaves honestly in Phase 1. $\mathcal{S}$ can encrypt 0 for each honest party in Flow 2.1, and simulate the partial decryptions for each honest party in Flow 2.3 using the partial decryption simulatability of threshold ElGamal.

Efficiency. Notice that a single execution of phase 1 suffices for multiple executions of phase 2, and that while in phase 1 the communication is $\Theta(n)$ per party, in phase 2 it is constant. So, the amortized communication complexity per party over multiple executions of phase 2 is constant.

Acknowledgements

We would like to thank Ran Canetti for helpful discussions.

References

AB06. C. Asmuth and J. Bloom. A modular approach to key safeguarding. *IEEE Trans. Inf. Theor.*, 29(2):208–210, September 2006.

ABG+13. Prabhanjan Ananth, Dan Boneh, Sanjam Garg, Amit Sahai, and Mark Zhandry. Differing-inputs obfuscation and applications. Cryptology ePrint Archive, Report 2013/689, 2013. `http://eprint.iacr.org/2013/689`.

BCNP04. Boaz Barak, Ran Canetti, Jesper Buus Nielsen, and Rafael Pass. Universally composable protocols with relaxed set-up assumptions. In *45th FOCS*, pages 186–195. IEEE Computer Society Press, October 2004.

Bd94. Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 274–285. Springer, Heidelberg, May 1994.

BGG+18. Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 565–596. Springer, Heidelberg, August 2018.

BGI+01. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18. Springer, Heidelberg, August 2001.

BGI+14a. Amos Beimel, Ariel Gabizon, Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, and Anat Paskin-Cherniavsky. Non-interactive secure multiparty computation. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 387–404. Springer, Heidelberg, August 2014.

BGI14b. Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, Heidelberg, March 2014.

BIK+17. Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 1175–1191. ACM Press, October / November 2017.

BJMS18. Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. Secure MPC: Laziness leads to GOD. Cryptology ePrint Archive, Report 2018/580, 2018. `https://eprint.iacr.org/2018/580`.

BW13. Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Heidelberg, December 2013.

BZ14. Dan Boneh and Mark Zhandry. Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 480–499. Springer, Heidelberg, August 2014.

CDPW07. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor,

TCC 2007, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.

CFY17. Robert K. Cunningham, Benjamin Fuller, and Sophia Yakoubov. Catching MPC cheaters: Identification and openability. In Junji Shikata, editor, *ICITS 17*, volume 10681 of *LNCS*, pages 110–134. Springer, Heidelberg, November / December 2017.

CR03. Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 265–281. Springer, Heidelberg, August 2003.

CS03. Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 126–144. Springer, Heidelberg, August 2003.

CSS12. T.-H. Hubert Chan, Elaine Shi, and Dawn Song. Privacy-preserving stream aggregation with fault tolerance. In Angelos D. Keromytis, editor, *FC 2012*, volume 7397 of *LNCS*, pages 200–214. Springer, Heidelberg, February / March 2012.

DF90. Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 307–315. Springer, Heidelberg, August 1990.

DHMR07. Vanesa Daza, Javier Herranz, Paz Morillo, and Carla Ràfols. CCA2-secure threshold broadcast encryption with shorter ciphertexts. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *ProvSec 2007*, volume 4784 of *LNCS*, pages 35–50. Springer, Heidelberg, November 2007.

DKSW09. Yevgeniy Dodis, Jonathan Katz, Adam Smith, and Shabsi Walfish. Composability and on-line deniability of authentication. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 146–162. Springer, Heidelberg, March 2009.

EDG14. Tariq Elahi, George Danezis, and Ian Goldberg. PrivEx: Private collection of traffic statistics for anonymous communication networks. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 1068–1079. ACM Press, November 2014.

ElG84. Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *CRYPTO'84*, volume 196 of *LNCS*, pages 10–18. Springer, Heidelberg, August 1984.

GM84. Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

GMR88. Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988.

GW09. Craig Gentry and Brent Waters. Adaptive security in broadcast encryption systems (with short ciphertexts). In Antoine Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 171–188. Springer, Heidelberg, April 2009.

KPTZ13. Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 669–684. ACM Press, November 2013.

Mer88. Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378. Springer, Heidelberg, August 1988.

MW16. Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 735–763. Springer, Heidelberg, May 2016.

RN10. Vibhor Rastogi and Suman Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 735–746. ACM, 2010.

Row05. J.K. Rowling. *Harry Potter and the Half-Blood Prince*. Bloomsbury, 2005.

RY16. Leonid Reyzin and Sophia Yakoubov. Efficient asynchronous accumulators for distributed PKI. In Vassilis Zikas and Roberto De Prisco, editors, *SCN 16*, volume 9841 of *LNCS*, pages 292–309. Springer, Heidelberg, August / September 2016.

SCR+11. Elaine Shi, T.-H. Hubert Chan, Eleanor G. Rieffel, Richard Chow, and Dawn Song. Privacy-preserving aggregation of time-series data. In *NDSS 2011*. The Internet Society, February 2011.

Sha79. Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

SW14. Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In David B. Shmoys, editor, *46th ACM STOC*, pages 475–484. ACM Press, May / June 2014.

Zha14. Mark Zhandry. Adaptively secure broadcast encryption with small system parameters. Cryptology ePrint Archive, Report 2014/757, 2014. `http://eprint.iacr.org/2014/757`.

A Threshold Encryption Scheme: Threshold ElGamal

One simple example of a (non ad hoc) threshold encryption scheme is the threshold ElGamal scheme TEG due to Desmedt and Frankel [DF90], described in Figure 6. TEG is defined over a group $\mathcal{G}$ of prime order p with generator g in which the decisional Diffie-Hellman problem is assumed to be hard.

Lemma 1. *Threshold ElGamal is (n, t)-statically secure (Definition 5, modified to use pk instead of $\{pk_i\}_{i\in\mathcal{U}}$) for any polynomial n, t as long as the Decisional Diffie-Hellman (DDH) assumption holds in $\mathcal{G}$.*

Informally, this lemma follows by a standard reduction from the DDH assumption.

Lemma 2. *Threshold ElGamal is (n, t)-partial decryption simulatable (Definition 4, modified to use pk instead of $\{pk_i\}_{i\in\mathcal{U}}$) as long as the Decisional Diffie-Hellman assumption holds in $\mathcal{G}$.*

Informally, this lemma follows since partial decryptions can easily be simulated by interpolation in the exponent.

$\mathsf{Setup}(1^k, t)$**:**
- Pick a secret key $sk \leftarrow^{\$} [p]$, and a random polynomial f of degree t with sk as its y-intercept.
- Return $pk = g^{sk}, \mathsf{msk} = f$.

$\mathsf{KeyGen}(\mathsf{msk})$**:**
- Pick a random $i \leftarrow^{\$} [1, \ldots, p-1]$.
- Return $sk_i = f(i)$.

$\mathsf{Enc}(pk, m \in \mathcal{G})$**:**
- Pick a random $y \in [p]$.
- $u = g^y$.
- $v = (pk)^y m$.
- Return $c = (u, v)$.

$\mathsf{PartDec}(sk_j, c = (u, v))$**:**
- Return $d_j = u^{sk_j}$.

$\mathsf{FinalDec}(\{d_i\}_{i \in \mathcal{R}' \subseteq \mathcal{R}}, c = (u, v))$**:**
- Interpolate the partial decryptions in the exponent to get u^{sk}: $y = \prod_{i \in \mathcal{R}' \subseteq \mathcal{R}} d_j^{\lambda_i}$, where λ_i is the appropriate Lagrange coefficient. (The Lagrange coefficients used depend on the identities i of the parties who participate. However, given that a certain threshold of parties do, the Lagrange coefficients do not affect the output.)
- Return $m = \frac{v}{y}$.

$\mathsf{Eval}(pk, c_1 = (u_1, v_1), c_2 = (u_2, c_2), \times)$**:**
- $u = u_1 u_2$
- $v = v_1 v_2$
- Return $c = (u, v)$.

Fig. 6: Threshold ElGamal Multiplicatively Homomorphic Encryption Scheme (TEG)

B Lower Bounds on Ciphertext Size for $\mathcal{R}$-Oblivious Ad Hoc Threshold Encryption Schemes

Theorem 8. *In any $\mathcal{R}$-flexible $\mathcal{R}$-oblivious ad hoc threshold encryption scheme* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{PartDec}, \mathsf{FinalDec})$*, the average size of a ciphertext c produced as*

$$(\mathsf{params}) \leftarrow \mathsf{Setup}(1^k, t = 1)$$

$$\{(pk_i, sk_i) \leftarrow \mathsf{KeyGen}(\mathsf{params})\}_{i \in [u]}$$

$$\mathcal{R} \leftarrow \text{ a random size-}n\text{ subset of } [u]$$

$$c \leftarrow \mathsf{Enc}(\mathsf{params}, \{pk_i\}_{i \in \mathcal{R}}, m)$$

for any k-bit message m is $O(\log_2 \binom{u}{n})$.

Proof. To see this, imagine that a challenger runs all four lines described above (that is, generates u key pairs, a random $\mathcal{R}$ and a ciphertext). The challenger then sends the key pairs to the adversary, whose task is to identify the keys belonging to $\mathcal{R}$. The challenger sends c to the adversary; the adversary attempts decryption with each key pair, and identifies those for which decryption yields m as belonging to $\mathcal{R}$. Note that the probability of correct decryption with a key that does not belong to $\mathcal{R}$ should be negligible, or the threshold encryption scheme is not secure. This allows the adversary to learn $\mathcal{R}$. Since there are $\binom{u}{n}$ possibilities for $\mathcal{R}$, it should require at least $\log_2 \binom{u}{n}$ bits to communicate. Since the key pairs are generated independently of $\mathcal{R}$, they don't count towards those bits; thus, the ciphertext should be at least $\log_2 \binom{u}{n}$ bits long. In the case when $u = 2n$, this is lower-bounded by 2^n.

Theorem 9. *For any $t < u$, any $\mathcal{R}$-flexible $\mathcal{R}$-oblivious ad hoc threshold encryption scheme* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{PartDec}, \mathsf{FinalDec})$*, the average size of a ciphertext c produced as*

$$(\mathsf{params}) \leftarrow \mathsf{Setup}(1^k, t)$$

$$\{(pk_i, sk_i) \leftarrow \mathsf{KeyGen}(\mathsf{params})\}_{i \in [u]}$$

$$\mathcal{R} \leftarrow \text{ a random size-}n\text{ subset of } [u]$$

$$c \leftarrow \mathsf{Enc}(\mathsf{params}, \{pk_i\}_{i \in \mathcal{R}}, m)$$

for any k-bit message m is $O(\log_2 \binom{u-t+1}{n})$.

Proof. The proof goes exactly as it does for Theorem 8, but the challenger withholds a random $t - 1$ key pairs in $\mathcal{R}$, so that the adversary only needs to identify the remaining $n - t + 1$ key pairs in $\mathcal{R}$. In order to help the adversary with this, the challenger additionally sends the adversary the partial decryptions generated by the withheld keys.

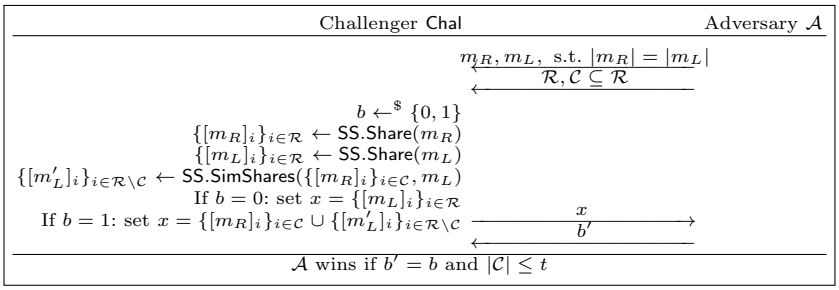

Fig. 7: Share Simulatability Game for Secret Sharing

C Background: Secret Sharing

Secret sharing was introduced by Shamir [Sha79]. A t-out-of-n sharing of a secret m is an encoding of the secret into n pieces, or *shares*, such that any $t+1$ shares together can be used to reconstruct the secret m, but t or fewer shares give no information at all about m. A secret sharing scheme SS, implicitly parametrized by the total number of shares n and the threshold t, consists of two algorithms: $\mathsf{SS.Share}$ and $\mathsf{SS.Reconstruct}$.

- $\mathsf{SS.Share}(m) \to ([m]_1, \ldots, [m]_n)$ takes in a secret m and produces the n secret shares.
- $\mathsf{SS.Reconstruct}([m]_{i_1}, \ldots, [m]_{i_{t+1}}) \to \tilde{m}$ takes in $t+1$ secret shares and returns the reconstructed secret $\tilde{m}$.

If the scheme does not require any setup that fixes n and t, we can include these parameters as inputs to $\mathsf{SS.Share}$.

Informally, correctness requires that $\tilde{m} = m$, and privacy requires that given t or fewer shares of either m_R or m_L, no efficient adversary can guess which message was shared.

Share Simulatability. We additionally use a property which we call *share simulatability*, which requires that given t or fewer honestly generated shares of m_R and given m_L, there exists an efficient algorithm $\mathsf{SS.SimShares}$ which generates the rest of the shares in such a way that the resulding sharing is indistinguishable from a fresh sharing of m_L.

Definition 6 (Share Simulatability). *A secret sharing scheme scheme* ($\mathsf{SS.Share}$, $\mathsf{SS.Reconstruct}$) *is* share simulatable *if there exists an there exists an efficient algorithm* $\mathsf{SS.SimShares}$ *such that no efficient adversary $\mathcal{A}$ can win the share simulatability game described in Figure 7 with probability non-negligibly greater than $\frac{1}{2}$.*

Shamir Secret Sharing [Sha79]. Shamir t-out-of-n secret sharing (Shamir) uses degree-(t) polynomials over some field. Shamir.Share(m) generates a random degree-(t) polynomial f with m as its y-intercept; each share $[m]_i$ is a point $(x_i, f(x_i))$ on the polynomial (with $x_i \neq 0$). Any $t+1$ shares can be used to interpolate the polynomial, reconstructing m. Any t or fewer shares give no information about m.

Shamir secret sharing is share simulatable; any t or fewer points can be interpolated with $(0, m_L)$ (and optionally with some additional random points) to obtain a degree-t polynomial.

Additionally, Shamir secret sharing is linearly homomorphic: a shared value m can be multiplied by a constant, or added to another shared value m', by separately operating on the individual shares.

D Proofs of Properties of the Share-and-Encrypt Ad Hoc Threshold Encryption Construction

In this appendix, we prove Theorem 1.

Theorem 10 (Restated from Theorem 1). *The share-and-encrypt ATE (Construction 1) is (n,t)-statically secure (Definition 5), as long as* SS *is a secure share simulatable t-out-of-n secret sharing scheme, and* PKE *is a CPA-secure public key encryption scheme.*

Theorem 1 claims that the share-and-encrypt ATE protocol is (n,t)-statically secure; in order to prove this, we must show that it is (n,t)-statically semantically secure and (n,t)-partial decryption simulatable.

D.1 Proof that Share-and-Encrypt is Statically Semantically Secure

In the proof below, we use a slightly non-standard version of the CPA security game (which we call the parallel CPA security game, described in Figure 8), where the challenger Chal operates multiple instances of the public key encryption scheme, and uses the same bit b for them all. Parallel CPA security is equivalent to CPA security, up to a polynomial difference in adversary advantage.

Proof. We show a sequence of indistinguishable games between a static security challenger Chal and an adversary $\mathcal{A}$. It starts with a challenger who always flips $b = R$, and ends with a challenger who always flips $b = L$. If the adversary cannot distinguish between games with those two challengers, then if we instead have a challenger who chooses $b \leftarrow^{\$} \{R, L\}$ uniformly at random (as in Figure 1), no adversary should be able to guess b with non-negligible advantage.

Game 0 This is the real game as described in Figure 1, but Chal always picks $b = R$ (and thus encrypts m_R).

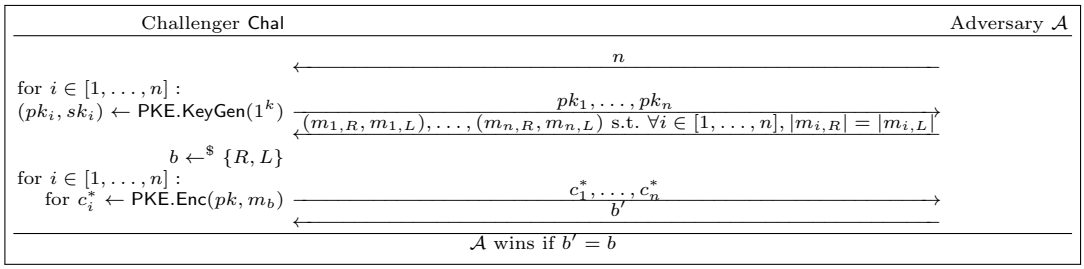

Fig. 8: Public Key Encryption Parallel CPA Security Game

Game 1 This game is the same as the previous game, but when computing the challenge ciphertext c^*, Chal does the following:

- $\{[m_L]_i\}_{i \in \mathcal{R}} \leftarrow \mathsf{SS.Share}(t, |\mathcal{R}|, m_L)$
- $\{[m_R]_i\}_{i \in \mathcal{R} \setminus \mathcal{C}} \leftarrow \mathsf{SS.SimShares}(\{[m_L]_i\}_{i \in \mathcal{R} \cap \mathcal{C}}, m_L)$
- $c^* \leftarrow \{\mathsf{PKE.Enc}(pk_i, [m_L]_i)\}_{i \in \mathcal{R} \cap \mathcal{C}} \cup \{[m_R]_i\}_{i \in \mathcal{R} \setminus \mathcal{C}}$

If $\mathcal{A}$ can tell the difference between this game and the previous game, then we can design another adversary $\mathcal{B}$ that uses $\mathcal{A}$ to break the share simulatability property of the secret sharing scheme SS (Definition 6). $\mathcal{B}$ accepts $\mathcal{A}$'s choice of $\mathcal{C}$, honestly generates all of the key pairs (like the challenger in Figure 1), and sends the appropriate keys to $\mathcal{A}$, who responds with $\mathcal{R}$, m_R and m_L. $\mathcal{B}$ aborts if $|\mathcal{C} \cap \mathcal{R}| > t$. ($\mathcal{B}$ continues with non-negligible probability, since when an abort happens $\mathcal{A}$ would have lost the game in Figure 1 anyway, and we're assuming that $\mathcal{A}$ wins it with non-negligible probability.) $\mathcal{B}$ forwards $(m_R, m_L, \mathcal{R}, \mathcal{R} \cap \mathcal{C})$ to the share simulatability challenger (Figure 7). Upon receiving shares from the share simulatability challenger, $\mathcal{B}$ encrypts them and sends them to $\mathcal{A}$. If the share simulatability challenger flips $b = 0$, $\mathcal{A}$'s view will be as in the previous game; if the share simulatability challenger flips $b = 1$, $\mathcal{A}$'s view will be as in this game. $\mathcal{B}$ sends the share simulatability challenger $b'' = 0$ if $\mathcal{A}$ submits $b' = R$, and $b'' = 1$ if $\mathcal{B}$ submits $b' = L$.

Game 2 This game is the same as the previous game, but when computing the challenge ciphertext c^*, Chal encrypts m_L.

If $\mathcal{A}$ can tell the difference between this game and the previous game, then we can design another adversary $\mathcal{B}$ that uses $\mathcal{A}$ to break the CPA security of the public key encryption scheme PKE. In particular, $\mathcal{B}$ will break parallel CPA security.

$\mathcal{B}$ does the following: after $\mathcal{A}$ specifies the corrupt set $\mathcal{C} \subseteq \mathcal{U}$, $\mathcal{B}$ honestly generates (pk_i, sk_i) for $i \in \mathcal{C}$, and talks to a parallel privacy PKE challenger (which runs $|\mathcal{U}| - |\mathcal{C}|$ instances of a PKE, as described in Figure 8) to get pk_i for $i \notin \mathcal{C}$. $\mathcal{B}$ sends $\mathcal{A}$ all of the public keys $\{pk_i\}_{i \in \mathcal{U}}$, and the corrupt secret keys $\{sk_i\}_{i \in \mathcal{C}}$. When $\mathcal{A}$ sends $\mathcal{B}$ the tuple $(\mathcal{R}, m_R, m_L)$, $\mathcal{B}$ aborts if $|\mathcal{C} \cap \mathcal{R}| > t$. ($\mathcal{B}$ continues with non-negligible probability, since when an abort happens $\mathcal{A}$ would have lost the game in Figure 1 anyway, and we're assuming that $\mathcal{A}$ wins it with non-negligible probability.) $\mathcal{B}$ then computes the following:

– $\{[m_L]_i\}_{i\in\mathcal{R}} \leftarrow \mathsf{SS.Share}(t, |\mathcal{R}|, m_L)$
– $\{[m_R]_i\}_{i\in\mathcal{R}\setminus\mathcal{C}} \leftarrow \mathsf{SS.SimShares}(\{[m_L]_i\}_{i\in\mathcal{R}\cap\mathcal{C}}, m_L)$

$\mathcal{B}$ computes $\{c_i^* \leftarrow \mathsf{PKE.Enc}(pk_i, [m_L]_i)\}_{i\in\mathcal{R}\cap\mathcal{C}}$. It then asks the PKE parallel CPA security challenger for encryptions of either $\{[m_R]_i\}_{i\in\mathcal{R}\setminus\mathcal{C}}$ or $\{[m_L]_i\}_{i\in\mathcal{R}\setminus\mathcal{C}}$ (depending on the parallel CPA security challenger's choice of bit b); let $\{c_i^*\}_{i\in\mathcal{R}\setminus\mathcal{C}}$ be the ciphertexts the parallel CPA security challenger returns. $\mathcal{B}$ sends $\mathcal{A}$ the challenge ciphertext $c^* = \{c_i^*\}_{i\in\mathcal{R}}$. If the parallel CPA security challenger chose $b = R$, $\mathcal{A}$'s view will be as in the previous game; if the parallel CPA security challenger chose $b = L$, $\mathcal{A}$'s view will be as in this game. $\mathcal{A}$ then sends $\mathcal{B}$ a guess b', which $\mathcal{B}$ passes on to the PKE parallel CPA security challenger. $\mathcal{B}$ wins the PKE parallel CPA security game exactly when $\mathcal{A}$ wins the static security game.

D.2 Proof that Share-and-Encrypt is Partial Decryption Simulatable

Proof. PartDecSim can simulate partial decryptions in the share-and-encrypt ad hoc threshold encryption scheme in Construction 1 simply by running $\{d_i\}_{i\in\mathcal{R}\setminus\mathcal{C}} \leftarrow \mathsf{SS.SimShares}(\{d_i\}_{i\in\mathcal{R}\cap\mathcal{C}}, m_R)$, and returning $\{d_i\}_{i\in\mathcal{R}\setminus\mathcal{C}}$.

Any adversary $\mathcal{A}$ who can win the static partial decryption simulatability game described in Figure 2 when played with PartDecSim with non-negligible probability can be used to break the share simulatability of SS and win the share simulatability game described in Figure 7.

E Share-and-Encrypt HATE Instantiations

In this appendix, we instantiate the share-and-encrypt HATE (Construction 1) in two ways.

E.1 Shamir-and-ElGamal

We build share-and-encrypt HATE out of ElGamal encryption [ElG84] and a variant of Shamir secret sharing. We need to use a *variant* of Shamir secret sharing (which we call exponential Shamir secret sharing), and not Shamir secret sharing itself, because Shamir secret sharing is additively homomorphic (and the homomorphism is applied via addition of individual shares), but ElGamal is multiplicatively homomorphic (and the homomorphism is applied via multiplication of ciphertexts), so if we attempt to apply a homomorphism on encrypted shares, it will not work. What we need in order to get an additively homomorphic ATE scheme is to use ElGamal encryption with a secret sharing scheme which is additively homomorphic, but whose homomorphism is applied via multiplication. Therefore, we need to alter our Shamir secret sharing scheme by moving the shares to the exponent; then, taking a product of two shares will result in a share of the sum of the two shared values. Below we describe the ElGamal encryption scheme and the exponential Shamir secret sharing scheme which we use.

ElGamal Multiplicatively Homomorphic Encryption. Figure 9 describes the ElGamal multiplicatively homomorphic encryption scheme (EG). Note that we split the key generation algorithm into two algorithms: Setup and KeyGen. This is because when we use ElGamal as part of our HATE scheme, it is important that all parties share the same modulus and generator, so we factor out part of KeyGen into Setup, which will only be run once globally.

Setup(1^k):
- Pick a prime-order group $\mathcal{G}$ with generator g in which the decisional Diffie-Hellman problem is assumed to be hard. Let p be the order of that group.
- Publish $\mathsf{params} = (\mathcal{G}, p, g)$.

KeyGen(params):
- Pick a random $sk \in [p]$.
- Publish $pk = g^{sk}$.

Enc(params, $pk, m \in \mathcal{G}$):
- Pick a random $y \in [p]$.
- $u = g^y$.
- $v = (pk)^y m$.
- Return $c = (u, v)$.

Dec(params, $sk, c = (u, v)$):
- Return $m = \frac{v}{u^{sk}}$.

Eval(params, $c_1 = (u_1, v_1), c_2 = (u_2, c_2), \times$):
- $u = u_1 u_2$.
- $v = v_1 v_2$.
- Return $c = (u, v)$.

Fig. 9: ElGamal Multiplicatively Homomorphic Public Key Encryption Scheme (EG) [ElG84]

Exponential Shamir Secret Sharing. Figure 10 describes the exponential Shamir secret sharing scheme (EShamir).

Notice that the reconstruction uses brute force search; this means that this secret sharing scheme can only be used for very small (polynomial-size in k) message spaces. However, HATE is interesting even in this setting. For instance, if all we want to do is take a poll by summing encryptions of 0s and 1s, this HATE scheme enables us to do it. It is reasonable to assume that the server can manage to do brute force search over a polynomial space, since it is already doing quadratic work in this computation.

Lemma 3. *The exponential Shamir secret sharing scheme (*EShamir*) described in Figure 10 is share simulatable.*

Proof. Informally, given a message m and t or fewer shares, we obtain correctly distributed remaining shares by interpolating the given with $(0, g^m)$ (and possibly with random values, if fewer than t shares are provided) in the exponent. This is done in a manner similar to the first step of reconstruction.

$\mathsf{Setup}(1^k)$: same as $\mathsf{EG.Setup}$.

$\mathsf{Share}(n, t \leq n, m \in \mathbb{Z}_p)$:

- Pick a random degree-t polynomial f in $\mathbb{Z}_p$ which has m as its y-intercept. This can be done by picking t random coefficients $\mathsf{coef}_1, \ldots, \mathsf{coef}_{t-1} \in \mathcal{G}$, and setting $f(x) = m + \sum_{j=1}^{t-1} \mathsf{coef}_j x^j$.
- Return $\{\mathsf{Share}_i\}_{i \in [1,\ldots n]}$ where $\mathsf{Share}_i = (i, g^{f(i)})$.

$\mathsf{Reconstruct}(\{\mathsf{Share}_i = (i, y_i)\}_{i \in \mathcal{R}' \subseteq [n]})$:

- Perform polynomial interpolation over the shares in the exponent to recover $g^{\tilde{m}}$. As long as $|\mathcal{R}'| > t$, this can be done by throwing out values in $\mathcal{R}'$ until $|\mathcal{R}'| = t + 1$, and doing the following:

$$g^{\tilde{m}} = \prod_{i \in \mathcal{R}'} y_i^{\prod_{j \in \mathcal{R}', j \neq i} \frac{j}{j-i}}$$

- Recover $\tilde{m}$ by brute force search.

$\mathsf{Eval}(\mathsf{Share}_i = (i, y_i), \mathsf{Share}'_i = (i, y'_i), +)$: $\mathsf{Share}^+_i = (i, y_i y'_i)$

Fig. 10: Exponential Shamir Secret Sharing Scheme ($\mathsf{EShamir}$)

E.2 CRT-and-Paillier

We also build share-and-encrypt HATE out of Camenisch-Shoup encryption and Chinese Remainder Theorem based secret sharing. Unlike Shamir-and-ElGamal (Section E.1), this HATE allows us to use large message spaces.

CS Additively Homomorphic Encryption. We use a slightly modified version of the Paillier-style verifiable encryption scheme described by Camenisch and Shoup [CS03].[7] Figure 11 describes the this scheme. Our modifications consist solely of removing elements from the ciphertext, so the modified scheme naturally inherits the CPA security of the original (but not its CCA security).[8].

CRT Secret Sharing. We use a classic secret sharing scheme based on the Chinese Remainder Theorem, which allows each party to operate homorphically on shares in a different group. This version is due to Asmuth and Bloom [AB06]. We describe it in Figure 12.

The scheme is perfectly correct. Furthermore, it supports a limited number (currently set to n) of homomorphic additions. The setting of parameters in the setup phase in Figure 12 ensures that $n \cdot A \leq N_+$, where each individual sharing corresponds to a vector of modular reductions of an integer less than A. This means that n sharings, added coordinate-wise, will lead to the reconstruction of an integer less than $n \cdot A$. Every set of more than t shares contains enough information for that reconstruction.

[7] Their scheme is designed it to be secure against chosen ciphertext attacks, which is unnecessary for our purposes.

[8] A similarly modified version of this scheme was used by Cunningham *et al.* [CFY17]

$\mathsf{KeyGen}(1^k)$:
- Let $N = pq$ where $p = 2p' + 1$ and $q = 2q' + 1$, and p' and q' are k-bit primes.
- Let $h = 1 + N$.
- Choose a random $g' \in Z^*_{N^2}$, and set $g = (g')^{2N} \bmod N^2$. (g is a generator of a size-$p'q'$ subgroup with high probability.)
- Choose a random secret key $sk \in \{1, \ldots, \lfloor (N^2)/4 \rfloor\}$.
- Return $pk = g^{sk} \bmod N^2$.

$\mathsf{Enc}(pk, m)$:
- Choose a random $r \in [N/4]$.
- Return $c = (g^r \bmod N^2, pk^r h^m \bmod N^2)$.

$\mathsf{Dec}(sk, c = (u, v))$:
- $z = \frac{v}{u^{sk}} \bmod N^2$. (Note that $z = h^m$ if c is an encryption of m.)
- Return $m' = \frac{z-1}{N}$ with division over integers. (Note that m' is the discrete log of z w.r.t. h.)

Fig. 11: Camenisch-Shoup Additively Homomorphic Encryption Scheme (CS). We omit Setup, since this scheme does not require setup.

The scheme's statistical security relies on the requirement that each unauthorized set of shares can reconstruct the secret integer $\hat{a}$ only modulo some integer that is (a) relatively prime to N_0 and (b) at most N_-, which itself is at most $\frac{A}{N_0 2^k}$. By the following lemma, those conditions ensure that the view of any unauthorized set is within statistical difference 2^{-k} of uniform.

Lemma 4. *Let a, n, t be positive integers, and let A be uniformly random in $\{a' \in [a] : a' \bmod n = t\}$. Then for all positive integers $m < a$ that are relatively prime to n, the distribution of the random variable $B = A \bmod m$ is within statistical difference $\frac{nm}{a}$ of uniform.*

Proof. Consider the number of $a' \in [a]$ that solve both the equations $a' \bmod n = t$ and $a' \bmod m = u$ (for some u). Since m and n are relatively prime, this system is equivalent to $a' \bmod mn = v$ for some particular v. The number of solutions to this is $\lfloor \frac{a}{mn} \rfloor$ or $\lceil \frac{a}{mn} \rceil$. Thus, the probability that $B = u$ is always with $1 \pm \frac{mn}{a}$ of a uniform element of $\mathbb{Z}_m$. The total variation distance from uniform is thus at most $\frac{mn}{a}$.

Lemma 5. *The CRT secret sharing scheme (CRTss) described in Figure 12 is share simulatable.*

Proof. Recall the share simulatability game from Figure 7. On input a set of unauthorized shares $\{\mathsf{Share}_i\}_{i \in \mathcal{C}}$ which were created as a sharing of m_L, and a target message m_R, first find a nonegative integer $a < N_0 \prod_{i \in \mathcal{C}} N_i$ such that $a \bmod N_0 = m_R$ and $a \bmod N_i = \mathsf{Share}_i$ for $i \in \mathcal{C}$. Such an integer exists since the moduli are all relatively prime. Next, select a random $\hat{a}_R \in [A]$ such that $\hat{a}_R \bmod (N_0 \prod_{i \in \mathcal{C}} N_i) = a$. The correctness condition of the secret sharing

scheme implies that $A > N_0 \prod_{i \in \mathcal{C}} N_i$, so this step is always possible. Finally, we produce the new shares as $\mathsf{Share}'_i = \hat{a} \bmod N_i$ for $i \in \mathcal{R} \setminus \mathcal{C}$.

By Lemma 4 above, the distribution of t or fewer shares of m_L are statistically indistinguishable from the corresponding distibution for m_R. The share simulation algorithm above selects a uniformly random sharing of m_R that is consistent with the unauthorized shares of m_L. The joint distribution is therefore statistically close to that of a fresh sharing of m_L.

$\mathsf{Share}(n, N_1, ..., N_n, N_0, t \leq n, m \in Z_{N_0}, 1^k)$:

- Let $\begin{cases} N_+ \stackrel{\text{def}}{=} \min_{J \subseteq [n] : |J| = t+1} \prod_{i \in J} N_i \\ A \stackrel{\text{def}}{=} \lfloor N_+ / n \rfloor \\ N_- \stackrel{\text{def}}{=} \max_{J \subseteq [n] : |J| = t} \prod_{i \in J} N_i \end{cases}$.
- If $N_0 \cdot 2^k \cdot N_- > A$ then stop and return "Error: message space too large for k bits of security."
- Select $a \in_R \{a' \in [A] : a' \bmod N_0 = m\}$
- Return $(\mathsf{Share}_1, \ldots, \mathsf{Share}_n)$ where $\mathsf{Share}_i = (i, a \bmod N_i)$.

$\mathsf{Reconstruct}(\mathsf{Share}_{i_1} = (i_1, y_{i_1}), \ldots, \mathsf{Share}_{i_t} = (i_t, y_{i_t}))$:

- Find the unique $\hat{a} \in Z_{\prod_j N_j}$ such that $\hat{a} \equiv y_i \mod N_i$ for all $i \in \{i_1, ..., i_t\}$.
- Return $\hat{m} = \hat{a} \bmod N_0$.

$\mathsf{Eval}(+, \mathsf{Share}_i = (i, y), \mathsf{Share}'_i = (i, y'))$: $\mathsf{Share}^+_i = (i, y + y' \bmod N_i)$

Fig. 12: Chinese Remainder Secret Sharing Scheme (CRTss). We omit Setup, since this scheme does not require setup.

F Security of the Obfuscation-Based Ad Hoc Threshold Encryption Construction

In order to make the threshold encryption definitions play nice with the obfuscation-based ATE (Construction 2), we need to alter the static semantic security game (and partial decryption simulatability game) in two ways.

First, we need to make the game even more static (what we call *super-static*) by forcing the adversary to commit not only to the set $\mathcal{C}$ of corrupt parties, but also the set $\mathcal{R} \cap \mathcal{C}$ of corrupt recipients. This is necessary for the proof of Theorem 4. Note that when $\mathcal{U} = \mathcal{R}$ (that is, all parties are recipients), super-static security is equivalent to static security.

Second, since Construction 2 is a keyed-sender scheme, we need to (a) use the sender secret key to encrypt and provide the adversary with the sender public key, and (b) we need to allow the adversary to make multiple encryption queries, since encryption is no longer a public operation. In typical public key encryption a game which allows multiple encryption queries is equivalent to one that does not, but this is not true in a keyed-sender setting.

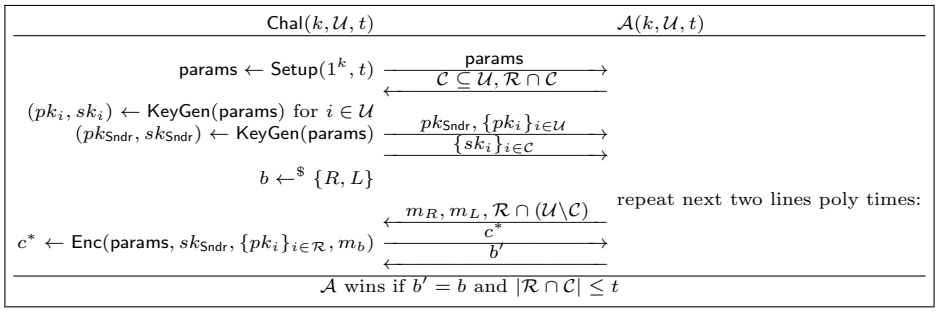

Fig. 13: Super-Static Semantic Security Game for Keyed-Sender Ad Hoc Threshold Encryption.

The new super-static semantic security game for keyed-sender ATE is described in Figure 13. (and the correspondingly modified super-static partial decryption simulatability game is described in Figure 14).

Definition 7 (super-static semantic security for keyed-sender ad hoc threshold encryption). *A keyed-sender ad hoc threshold encryption scheme* (Setup, KeyGen, Enc, PartDec, FinalDec) *is* (n, t)-super-statically semantically secure *if for all sufficiently large security parameters k, no efficient adversary $\mathcal{A}$ can win the game described in Figure 13 (with polynomial-size $\mathcal{U}$ and $|\mathcal{R}| = n$) with probability non-negligibly greater than $\frac{1}{2}$.*

Definition 8 (super-static partial decryption simulatability for keyed-sender ad hoc threshold encryption). *A keyed-sender ad hoc threshold encryption scheme* (Setup, KeyGen, Enc, PartDec, FinalDec) *is* (n, t)-super-statically partial decryption simulatable *if there exists an efficient algorithm* PartDecSim *such that for all sufficiently large security parameters k, no efficient adversary $\mathcal{A}$ can win the game described in Figure 14 (with polynomial-size $\mathcal{U}$ and $|\mathcal{R}| = n$) with probability non-negligibly greater than $\frac{1}{2}$.*

Definition 9 (super-static security for keyed-sender ad hoc threshold encryption). *A keyed-sender ad hoc threshold encryption scheme* (Setup, KeyGen, Enc, PartDec, FinalDec) *is* (n, t)-super-statically secure *if it is both* (n, t)-super-statically semantically secure *(Definition 7) and* (n, t)-super-statically partial decryption simulatable *(Definition 8).*

Theorem 11 (Restated from Theorem 4). *The obfuscation-based ATE (Construction 2) is (n, t)-super-statically secure (Definition 9) for any polynomial n, t, as long as* diO *is a secure differing-inputs obfuscation,* PPRF *is a secure puncturable* PRF, SIG *is an existentially unforgeable signature scheme,* PKE *is a CPA-secure public key encryption scheme, and the accumulator scheme is secure.*

In order to prove Theorem 4, we must show the obfuscation-based Homomorphic Ad Hoc Threshold Encryption construction is super-statically semantically secure and super-statically partial decryption simulatable.

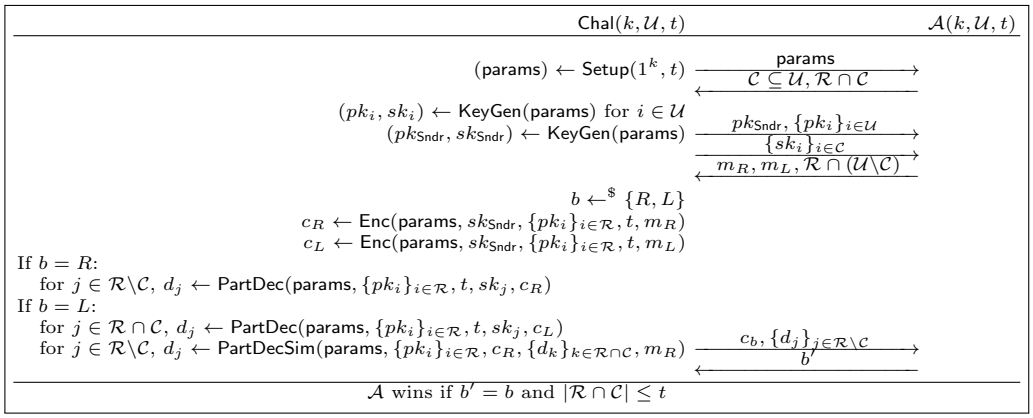

Fig. 14: Super-Static Partial Decryption Simulatability Game for Keyed-Sender Ad Hoc Threshold Encryption

F.1 Proof that Obfuscation-Based Homomorphic Ad Hoc Threshold Encryption Share-and-Encrypt is Super-Statically Semantically Secure

Proof. For simplicity, we first consider a single pair of messages m_R, m_L.

We show a sequence of indistinguishable games between a super-static semantic security challenger Chal and an adversary $\mathcal{A}$. It starts with a challenger who always flips $b = R$, and ends with a challenger who always flips $b = L$. If the adversary cannot distinguish between games with those two challengers, then if we instead have a challenger who chooses $b \leftarrow^{\$} \{R, L\}$ uniformly at random (as in Figure 13), no adversary should be able to guess b with non-negligible advantage.

In the sequence of games below, let $c^* = (c_1^*, c_2^*)$ denote the "one time pad" encryption of the message and r^* denote the random value embedded in the challenge ciphertext. Notice that even when the message changes, c_1^* and r^* stay constant (assuming the challenger uses the same randomness).

We abuse puncturable PRF notation slightly below; $\mathsf{PPRF}_k(x)$ denotes the puncturable PRF evaluated with key k on input x, while we use $\mathsf{PPRF}_{k,x^*,y^*}(x)$ to denote the same PPRF evaluated with the same key k, but if it is called on x^*, it is programmed to return the hardcoded value y^*, as defined in Algorithm 2 below.

Algorithm 2 $\mathsf{PPRF}_{k,x^*,y^*}(x)$

```
if x = x* then
    return y*
else
    return PPRF_k(x)
```

We also let $k\{x^*\}$ denote a PPRF key k punctured at x^*.

Game 0 This is the real game as described in Figure 13, but Chal always picks $b = R$ (and thus encrypts $m^* = m_R$).

Game 1 In this game, when creating the sender's public key (specifically, the obfuscation of Algorithm 1 it contains), the challenger Chal punctures the PPRF keys $k_1, \ldots, k_t$ at r^*. To preserve the input-output behavior of the program, Chal then modifies the program to check whether the input to PPRF_{k_j} is r^*, and if it is, to set $\mathsf{coef}_j = \mathsf{PRF}_{k_j}(r^*)$ anyway for all j. (That is, Chal sets $\mathsf{coef}_j = \mathsf{PPRF}_{k_j\{r^*\},r^*,\mathsf{PPRF}_{k_j}(r^*)}(r)$).
This game is indistinguishable from Game 0 by the properties of indistinguishable obfuscation; the two programs have identical input-output behavior.

Game 2 In this game, the challenger Chal modifies the obfuscated program to set the coefficients truly at random. That is, Chal chooses t random values $r_1, \ldots, r_t$, and sets the program to compute the coefficients as $\mathsf{coef}_j = \mathsf{PPRF}_{k_j\{r^*\},r^*,r_j}(r)$ for all j.
This game is indistinguishable from Game 1 by the security of puncturable PRFs.

Game 3 In this game, the challenger Chal punctures the PPRF key k' at c_1^*. Chal then modifies the program to check whether $c_1 = c_1^*$, and if it is, to set $w = \mathsf{PPRF}_{\mathsf{PPRF}.k'}(c_1^*)$ anyway (that is, Chal sets $w = \mathsf{PPRF}_{k'\{c_1^*\},c_1^*,\mathsf{PPRF}_{k'}(c_1^*)}(c_1)$).
This game is indistinguishable from Game 2 by the properties of indistinguishable obfuscation; the two programs have identical input-output behavior.

Game 4 Now, Chal modifies the program to set w to be uniformly random if $c_1 = c_1^*$. That is, Chal chooses a random value r', and sets $w = \mathsf{PPRF}_{k'\{c_1^*\},c_1^*,r'}(c_1))$ in the obfuscated program.
However, this alters the observable program behavior, since the message is now computed as $m' = c_2 \oplus \mathsf{PPRF}_{k'\{c_1^*\},c_1^*,r'}(c_1)$, which, on the challenge ciphertext, is equal to $m_R \oplus \mathsf{PPRF}_{k'}(c_1^*) \oplus r'$; to prevent this, at the same time, Chal programs the PPRF, when used with the punctured keys $k_1\{r^*\}, \ldots, k_t\{r^*\}$, to return the necessary coefficients so that the output of the program on the challenge ciphertext (and corrupt $i \in \mathcal{C} \cap \mathcal{R}$) does not change. Chal uses Algorithm 3 by calling $\mathsf{coef}' = (\mathsf{coef}'_1, \ldots, \mathsf{coef}'_t) = \mathsf{ChooseCoeffs}(\mathsf{coef} = (\mathsf{coef}_1, \ldots, \mathsf{coef}_t), \mathcal{C} \cap \mathcal{R}, \mathsf{PPRF}_{k'}(c_1^*) \oplus r')$ to do this. Chal modifies the program to compute coefficients as $\mathsf{coef}_j = \mathsf{PPRF}_{k_j,r^*,\mathsf{coef}'_j}(r)$.

Algorithm 3 $\mathsf{ChooseCoeffs}(\mathsf{coef} = (\mathsf{coef}_1, \ldots, \mathsf{coef}_t), x = (x_1, \ldots, x_t), \Delta m)$

Informally, compute $y_j = \sum_{j \in [1,\ldots,t]} \mathsf{coef}_j x_j^j$ for $j \in [t]$, and interpolate $(0, \Delta m), (x_1, y_1), \ldots, (x_t, y_t)$. Return the coefficients of this interpolated polynomial.

This game is indistinguishable from Game 3 by the properties of differing-inputs obfuscation; the two programs have identical input-output behavior

on all points which the adversary can find. If the adversary finds an input on which the two programs differ, then we can use that adversary to break either the public key encryption scheme, the accumulator scheme, or the signature scheme. The reasoning is as follows: if the adversary can find such an input, then it is for $i \notin \mathcal{C} \cap \mathcal{R}$ or for a different pair (a, c, r) tuple. Consider the following cases, which cover all possibilities:

- If a in the adversarial input is equal to the one in the challenge ciphertext and $i \notin \mathcal{R}$, then the adversary can break the accumulator security, because the accumulator was built for $\mathcal{R}$.
- If a in the adversarial input is equal to the one in the challenge ciphertext, $i \notin \mathcal{C}$ but $i \in \mathcal{R}$, then the adversary can break the one-wayness of key generation for the public-key encryption scheme, because the adversary has to find a secret key that matches an honest recipient's public key.
- If a, c or r in the adversarial input is not equal to the one in the challenge ciphertext, then the adversary can break the security of the signature scheme.

Game 5 Now Chal chooses a second random value r'', and sets $c_2^* = r''$. This game is indistinguishable from Game 1 by the security of puncturable PRFs. At this point, nothing depends on m_R.

The rest of the games are what we did before, but in reverse, with m_L instead of m_R.

Of course, the above proof only considered a single message, for simplicity. Informally, in order to move to multiple rounds of chosen plaintexts (as in the game in Figure 13), we do a sequence of hybrids, dealing with one message (and thus one pair of locations (c_1^*, r) at which to puncture) at at a time.

F.2 Proof that Obfuscation-Based Homomorphic Ad Hoc Threshold Encryption Share-and-Encrypt is Super-Partial Decryption Simulatable

PartDecSim is simply the Shamir secret sharing SimShares algorithm. The proof is very similar to the proof of super-static semantic security; we do not reproduce the hybrids here.

G Additively Server-Aided Homomorphic Obfuscation-Based HATE

In this appendix, we describe the additively server-aided homomorphic obfuscation-based HATE scheme. The program each sender must obfuscate and include in their public key is described in Algorithm 4. Notice that this program is now $O(n)$ instead of $O(t)$ in size. The obfuscation-based HATE is described in Construction 5.

We do not restate the proof of super-static semantic security; we only highlight the major differences from the proof in Appendix F.

Informally, in Game 4, Chal also hard-codes the ciphertexts returned for honest parties in order to preserve the program's input-output behavior.

We then add Games 4a, 4b and 4c.

In Game 4a, the PPRF keys $k'_1, \ldots, k'_n$ (which provide randomness for the encryption) are punctured at r. Since the ciphertexts are already hard-coded, the input-output behavior of the program does not change, and Game 4a is indistinguishable from Game 4 by the properties of indistinguishable obfuscation.

In Game 4b, Chal uses true randomness to compute the hard-coded ciphertexts. Game 4b is indistinguishable from Game 4a by the security of PPRF.

In Game 4c, Chal encrypts random messages to get the hard-coded ciphertexts. Game 4c is indistinguishable from Game 4b by the CPA security of the encryption scheme.

Game 5 is unchanged, and at this point everything is message-independent; we can reverse the games as before.

Algorithm 4 $f'_{k,k',k'',\mathsf{SIG}.pk}(a, c', r, \sigma, \mathsf{EG}.pk, i, w)$

The following values are hardcoded in the program:

- puncturable PPRF keys $k = (k_1, \ldots, k_t)$ {These are for generating polynomial coefficients.}
- puncturable PPRF key k' {This is for decrypting the message.}
- puncturable PPRF keys $k'' = (k''_1, \ldots, k''_n)$ {These are for encrypting shares.}
- signature verification key $\mathsf{SIG}.pk$

The following values are expected as input:

- accumulator a
- ciphertext $c' = (c_1, c_2)$
- random value r
- signature σ
- public encryption key $\mathsf{EG}.pk$
- index i
- witness w

if $(\mathsf{verify}(a, w, (\mathsf{EG}.pk, i)))$ and $(\mathsf{SIG.Verify}(\mathsf{SIG}.pk, (a, c', r), \sigma))$ **then**
  $(c_1, c_2) = c'$
  $w = \mathsf{PPRF}_{k'}(c_1)$
  $m = w \oplus c_2$
  for $j \in [1, \ldots, t]$ **do**
    $\mathsf{coef}_j = \mathsf{PPRF}_{k_j}(r)$
  for $i \in [1, \ldots, n]$ **do**
    $r_i = \mathsf{PPRF}_{k''_i}(r)$
  return $g^{\mathsf{EG.Enc}(\mathsf{EG}.pk, m + \sum_{j \in [1,\ldots,t]} \mathsf{coef}_j i^j; r_i)}$
  {This returns an encryption of the ith exponential Shamir share of m, $[m]_i$. Encryption uses randomness r_i. }

$\mathsf{Setup}(1^k)$: $\mathsf{params} \leftarrow \mathsf{EG.Setup}(1^k)$

$\mathsf{KeyGen}(\mathsf{params}, t)$:

This is exactly as in Construction 2, except that the sender generates n additional PPRF keys $k''_1, \ldots, k''_n$, and instead of obfuscating f from Algorithm 1 to get $\mathsf{ObfFunc}$, the sender obfuscates f' from Algorithm 4.

$\mathsf{Enc}(\mathsf{params}, \{\mathsf{EG}.pk_i\}_{i \in \mathcal{R}, |\mathcal{R}| \geq t}, m)$:

This is exactly as in Construction 2.

$\mathsf{PartDec}(\mathsf{params}, \mathsf{ObfFunc}_{\mathsf{Sndr}}, \{\mathsf{EG}.pk_j\}_{j \in \mathcal{R}}, \mathsf{EG}.sk_i, c)$:

if the ciphertext c is an output of a homomorphic evaluation **then**
$e = c$.
else
Parse $(a, c', r, \sigma) = c$.
Recompute a deterministic accumulator a (e.g. a Merkle hash tree) of $\{(\mathsf{PKE}.pk_j, j)\}_{j \in \mathcal{R}}$ (where the indices j are the indices of the public keys in a lexicographic ordering of all the public keys belonging to recipients in $\mathcal{R}$). Let w be the witness of $(\mathsf{PKE}.pk_i, i)$ in that accumulator.
$e = \mathsf{ObfFunc}_{\mathsf{Sndr}}(a, c', r, \sigma, \mathsf{EG}.pk_i, i, w)$.
$[m]_i \leftarrow \mathsf{EG.Dec}(\mathsf{EG}.sk_i, e)$
$d_i = (i, [m]_i)$.
return d_i.

$\mathsf{FinalDec}(\mathsf{params}, \{d_i\}_{i \in \mathcal{R}' \subset \mathcal{R}})$:

Perform exponential Shamir reconstruction $\mathsf{EShamir.Reconstruct}(\{d_i\}_{i \in \mathcal{R}'})$ as described in Figure 10 to recover m.

$\mathsf{Eval}(\mathsf{params}, \{pk_{\mathsf{Sndr}}\}_{\mathsf{Sndr} \in \mathcal{S}}, \{pk_i\}_{i \in \mathcal{R}}, [c_1, \ldots, c_l], +)$:

{Note that this algorithm receives the public keys for all senders $\mathsf{Sndr} \in \mathcal{S}$ (and thus their obfuscated programs). Without loss of generality, let c_q be from P_q (and therefore requiring the use of $\mathsf{ObfFunc}_q$).}
for $q \in [1, \ldots, l]$ **do**
Parse $(a_q, c'_q, r_q, \sigma_q) = c_q$.
for $i \in \mathcal{R}$ **do**
Recompute a deterministic accumulator a (e.g. a Merkle hash tree) of $\{(\mathsf{PKE}.pk_j, j)\}_{j \in \mathcal{R}}$ (where the indices j are the indices of the public keys in a lexicographic ordering of all the public keys belonging to recipients in $\mathcal{R}$). Let w_i be the witness of $(\mathsf{PKE}.pk_i, i)$ in that accumulator.
for $q \in [1, \ldots, l]$ **do**
if $a_q \neq a$ **then**
Abort.
$c_{i,q} \leftarrow \mathsf{ObfFunc}_q(a, c'_q, r_q, \sigma_q, \mathsf{EG}.pk_i, i, w_i))$.
$c^*_i = \mathsf{EG.Eval}(\mathsf{params}, \mathsf{EG}.pk_i, [c_{i,1}, \ldots, c_{i,l}], +)$
return $c^* = \{c^*_i\}_{i \in \mathcal{R}}$

Construction 5: Obfuscation-Based HATE