Secure Computation on the Web: Computing without Simultaneous Interaction

Shai Halevi Yehuda Lindell Benny Pinkas

April 27, 2011

Abstract

Secure computation enables mutually suspicious parties to compute a joint function of their private inputs while providing strong security guarantees. Amongst other things, even if some of the participants are corrupted the output is still correctly computed, and parties do not learn anything about each other's inputs except for that output. Despite the power and generality of secure computation, its use in practice seems limited. We argue that one of the reasons for this is that the model of computation on the web is not suited to the type of communication patterns needed for secure computation. Specifically, in most web scenarios clients independently connect to servers, interact with them and then leave. This rules out the use of secure computation protocols that require that *all* participants interact simultaneously.

In this paper, we initiate the study of secure computation in a client-server model where each client connects to the server *once* and interacts with it, without any other client necessarily being connected at the same time. We point out some inherent limitations in this model and present definitions that capture what can be done. We also present a general feasibility result and several truly practical protocols for a number of functions of interest. All our protocols are based on standard assumptions, and we achieve security both in the semi-honest and malicious adversary models.

Contents

1 Introduction **1**
1.1 Our Contributions . . . 2
1.1.1 Good Decompositions . . . 3
1.1.2 Securely Computing any Decomposition . . . 3
1.2 Some Related Work . . . 4

2 One-Pass Decompositions **4**
2.1 Minimum-Disclosure Decompositions . . . 5
2.2 Examples of Functions with Minimum-Disclosure Decompositions . . . 6

3 Server-Based One-Pass Protocols **8**

4 Practical Optimal Protocols **9**
4.1 Protocols for Symmetric Functions . . . 9
4.1.1 The Semi-Honest Case . . . 10
4.1.2 The Malicious Case . . . 13
4.2 Symmetric Functions over $\mathbb{Z}_{\boldsymbol{c}}$ . . . 16
4.3 A Protocol for The Sum Function . . . 17
4.4 The Selection Functions . . . 18

5 Securely Computing any Decomposition **20**
5.1 The GHV Construction . . . 21
5.2 Our Construction, Semi-Honest Model . . . 22
5.2.1 The Protocol . . . 23
5.3 Security in the Semi-Honest Model . . . 24
5.4 The Malicious Model . . . 26

6 Extensions and Open Problems **26**

References **27**

1 Introduction

Web-servers are a dominant communication medium in today's society. Some examples include users of social networks that communicate by sending messages to the web-servers of their network to "write on the wall" of their friends (and these servers distribute the messages to the intended recipients), program committees that use web-based systems to share their reviews and discussions, readers that participate in on-line polls on newspapers' web-sites, voters using web-based election systems, and so on.

The systems in all of these examples exhibit a star-like communication pattern: End-users never communicate directly, instead they send their messages to central web-servers, and these servers take care of processing the messages and/or forwarding them as needed. In many cases, direct interaction between users is impossible simply because users are off line most of the time. For example, readers may spend only a few minutes every day browsing the newspaper's web-site and participating in on-line polls. Similarly, program committee members may login to the review site every day or two to participate in the discussion and votes, and then log off until the next time.

In almost all systems today, the web-server serves not only as a communication medium but also as a trusted party. It receives all the information from the users and does all the processing, and it is trusted by the users to only use their information as needed for the application (or as specified in the "privacy policy" of the web site). This may be appropriate in some cases, but there are many cases where there is no reason for users to trust the server or each other, and indeed many cases where this trust was found to be unjustified in retrospect. For a few examples, see [20, 1, 8].

A natural approach toward rectifying this problem is to use some of our cryptography technology for eliminating trusted parties. Indeed, the last three decades saw a very significant body of work within the cryptography research community (going under the general name of *secure multi-party computation*), devoted to finding various ways of transforming systems that rely on trusted parties into systems that do not need them (see, e.g., [12, Ch. 7] for an overview).

In fact, with client-side processing in Web-2.0 we now have a huge mass of parties with serious computing platforms and conflicting interests, all wishing to interact with each other to perform some joint tasks. This seems to offer the perfect setting for mass deployment of secure multi-party computation, but in reality there are several reasons why such mass deployment does not happen. Some of these reasons are related to practical issues with browser technology (e.g., clients cannot verify that they run the right program). In this work, however, we consider a more cryptographic reason, namely the fact that our current multi-party protocols seem incompatible with the communication patterns in today's web applications. Much of the work on secure multi-party computation assumes that all parties remain on-line throughout the computation, and most solutions also rely on very strong communication primitives like secure broadcast. The question arises, then, whether we can eliminate the need for the web-server being a trusted party, even in this setting of loosely connected parties that are off line most of the time. Addressing this question is the focus of the current work.

We stress that beyond the practical interest that we discussed above, addressing multi-party computation in this model is also of significant theoretical interest. It is not at all clear that theoretically meaningful secure computation can be achieved in a setting where each party carries out a single interaction with an untrusted server at a different time (either in the semi-honest or the malicious settings). The power of this model is therefore a natural theoretical question to consider.

We note at the outset that a naive approach using fully homomorphic encryption [22, 9] does not solve the problem of secure computation in our setting: Although each party can encrypt its

input and the computation can be done homomorphically, there is still the need to decrypt the final ciphertexts while preventing decryption of the intermediate ciphertexts.

1.1 Our Contributions

In this paper we initiate a study of this scenario. We define security, and observe that in this setting it is not always possible to achieve the same level of security as in the standard setting of secure computation. We formalize what can be done in this model, and then present theoretical and practical constructions, for both the cases of semi-honest and malicious adversaries. Our constructions all rely on standard assumptions (like the DDH assumption) and are in the standard model. The only exception is that for our practical construction in the case of malicious adversaries, we use random oracles in order to obtain practical non-interactive zero-knowledge via the Fiat-Shamir paradigm [7].

We begin by considering a very basic setting of a server and n parties, denoted $P_1, P_2, \ldots, P_n$. Each party P_i has an input x_i, and the parties wish to jointly evaluate a function $f(x_1, \ldots, x_n)$ (e.g., the sum of the inputs, or their maximum value), such that the server learns the output value. To simplify the exposition, consider the case where the parties talk to the server in order, first party P_1, then party P_2, all the way up to party P_n, and if everyone cooperates then after talking to them all the server should be able to learn the output value.

Consider first the case of semi-honest parties. It is easy to see that protocols in this model *cannot* always provide the same privacy guarantees as standard secure-function-evaluation protocols (SFE). For example, if the last $n - i$ parties collude with the server, then they can always evaluate the residual function

$$g_i^{\vec{x}}(z_{i+1}, \ldots, z_n) \stackrel{\text{def}}{=} f(x_1, \ldots, x_i, z_{i+1}, \ldots, z_n)$$

on as many inputs $(z_{i+1}, \ldots, z_n)$ as they like. This is due to the fact that these last $n - i$ parties must have the capability of computing $f(x_1, \ldots, x_i, x_{i+1}, \ldots, x_n)$ for every possible vector of their inputs $x_{i+1}, \ldots, x_n$. Furthermore, since the first i parties are no longer involved, nothing prevents the last $n - i$ parties from just rerunning the rest of the protocol many times with different inputs $z_{i+1}, \ldots, z_n$.

We formalize the inherent "leakage" in this model by introducing the concept of a *one-way decomposition* of a function: A decomposition of an n-input function $f(x_1, x_2, \ldots, x_n)$ is a vector of functions $\{f_i(y_{i-1}, x_i) : i = 1, \ldots, n\}$, where y_i represents the *intermediate result* after taking in the inputs of parties 1 through i and y_0 is defined as the empty string, such that for all inputs $x_1, x_2, \ldots, x_n$ it holds that

$$f(x_1, x_2, \ldots, x_n) \ = \ f_n(\cdots f_2(f_1(x_1), x_2) \ldots, x_n).$$

One can see that every protocol for computing f in our model corresponds to some (possibly randomized) decomposition of f, roughly because we can think of y_i as the state of the server after interacting with party P_i. However, as we will see, not all decompositions are equal, some are better than others (and some are incomparable). We therefore break up the problem of secure computation in this model into (a) finding a "good" decomposition of the given function f, and (b) devising a protocol to securely compute a given decomposition.

This approach separates the inherent information leakage in our model from anything additional that may be learned from the computational process. This is analogous to the standard definition of secure computation that guarantees that the only information that can be learned about the

parties' private inputs is that which can be learned from the specified output. The question of which functions can be safely computed (i.e., for which the output does not reveal too much) is orthogonal to the question of how to compute the function so that only the output is revealed. Here too, the question of which decomposition can be safely computed is orthogonal to the question of how to compute the function so that only the inherent leakage due to the setting and decomposition is learned.

1.1.1 Good Decompositions

Although every function f can be decomposed as described above, some decompositions are more "interesting" or "natural" than others. A trivial example is that any function f can be decomposed by setting the functions $f_1, \ldots, f_{n-1}$ to all be the identity function and then setting $f_n = f$. A more interesting example is that the sum function, $f(x_1, \ldots, x_n) = \sum_i x_i$, can be decomposed by letting the f_i's be the partial sums, $f_i(y_{i-1}, x_i) = y_{i-1} + x_i$. Clearly, the decomposition of the sum function using partial sums is much better than its decomposition using the identity functions, since it reveals much less information to the adversary.

We are particularly interested in "*minimum-disclosure*" decompositions of f, where $y_i = f_i(\cdots)$ carries no more information about the inputs $x_1, \ldots, x_i$ than the truth-table of the residual function $g_i^{\vec{x}}$ from above. (i.e., the output of $g_i^{\vec{x}}$ on every possible $(z_{i+1}, \ldots, z_n)$ which can always be learned if the last $n-i$ parties collude with the server). It is easy to see that for the the sum function, having the f_i's be the partial sums is indeed a minimum-disclosure decomposition (see Section 4.3). In Section 2 we define this notion of minimum-disclosure decompositions and describe many functions that have efficient minimum-disclosure decompositions, and in Section 4.1 we describe practical protocols for securely computing some of these decompositions (in a PKI model). The functions that we can handle in this fashion include all the symmetric functions on small domains (and also some other functions), so for example we get a practical protocol for computing the majority function (or a referendum), as privately as can be in our model.

1.1.2 Securely Computing any Decomposition

Given a specific decomposition of f (that codifies the "leakage" that we are willing to tolerate while computing f in our model), what does it mean for a protocol to securely compute this decomposition? In keeping with the intuition that y_i represents the partial result up to party i, we set out to formalize the requirement that these partial results are the only thing that can be learned by the bad parties.

First, observe that many of the intermediate results y_i's can be hidden from the corrupted parties. For example, if parties P_1, P_2 and P_3 are honest then we expect the partial results y_1 and y_2 to remain hidden, even if a dishonest P_4 learns y_3. In fact our formal definition requires a little more: A protocol is said to securely compute a given decomposition of f if the only partial result that it leaks is the one after *the last honest party*. Namely, the view of any set of adversarial parties can be simulated knowing only the value $y_i = f_i(\ldots)$, where i is the index of the last honest party. Furthermore, if the server is honest, then nothing but the output of f is revealed.

The rationale for this definition is that as long as some honest parties are still on deck, we can hope to have some secrecy against the adversarial parties so we do not let them learn the current y_j's. However, once all the honest parties have had their turn and only the adversarial parties are left, we can no longer expect to have any secrecy so we let the adversarial parties learn that last y_i.

(We remark that a weaker definition that allows the bad parties to learn all the y_i's for which party $i+1$ is dishonest, is essentially equivalent to the notion of i-Hop homomorphic encryption from [11].)

In Section 5 we consider the task of devising a protocol to securely compute a particular given decomposition of a function f. Using re-randomizable garbled circuits similar to Gentry et al. [11] we show that under the DDH assumption any efficient decomposition of f can be securely computed in our model (if PKI is available). Our treatment simplifies the techniques from [11], in that we use re-randomizable garbled circuits only in conjunction with re-randomizable encryption (whereas [11] needed also re-randomizable OT). We also strengthen the construction from [11] slightly in order to deal with malicious parties. See Section 5 for more details about these points.

1.2 Some Related Work

Some of the techniques that we use for our practical protocols are similar to those used in the work of Harnik et al. [14]: In that work they considered a multi-party-computation settings where you incorporate the inputs of parties one at a time, with the goal of minimizing the number of OTs that are needed every time a new input is incorporated. In particular our protocols for symmetric functions are reminiscent of their Tables Method.

Another related work is that of Choi et al. [4]: They considered a setting where the parties can interact in a setup phase before receiving their inputs, and then they want to minimize online communication while maintaining full security. Their results are not applicable in our model, however, since, as we explained, full security cannot obtained in our model (and this remains true even given an interactive setup phase).

2 One-Pass Decompositions

Throughout the text we denote the number of parties (not counting the server) by n, and the security parameter by m. For an integer n we denote $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ and $[n] = \{1, 2, \ldots, n\}$. In the text we also refer to randomized functions which can be viewed as distributions over deterministic functions all with the same domain and range It it convenient to consider a randomized function $f : D \to R$ as one that takes as input $x \in D$ and randomness which is chosen from some well-specified distribution and outputs $y \in R$. A concatenation of randomized functions implies choosing the randomness for each one independently of the others.

Definition 2.1 (Decomposition). *Let $f : D^n \to R$ be an n-variable function (from domain D to range R). A deterministic* one-pass decomposition *of f is a sequence of functions $f_1 : D \to \{0,1\}^*$, $f_i : \{0,1\}^* \times D \to \{0,1\}^*$ for $i = 2, 3, \ldots, n-1$, and $f_n : \{0,1\}^* \times D \to R$ such that for all $x_1, \ldots, x_n \in D$, it holds that*

$$f(x_1, x_2, \ldots, x_n) \;=\; f_n(\cdots f_2(f_1(x_1), x_2) \cdots, x_n). \tag{1}$$

A randomized one-pass decomposition of f is a sequence of n randomized function with the same domains and ranges as above, such that Equation (1) holds with overwhelming probability (in the implicit security parameter).

Below we will omit the "one-pass" qualifier and just call this sequence of functions a decomposition. We often also omit the distinction between deterministic and randomized decompositions.

Given a decomposition $\bar{f} = \langle f_1, \ldots, f_n \rangle$, we denote by $\tilde{f}_i$ the concatenation of the first i functions,

$$\tilde{f}_i(x_1, x_2, \ldots, x_i) \stackrel{\text{def}}{=} f_i(\cdots f_2(f_1(x_1), x_2) \cdots, x_i). \tag{2}$$

2.1 Minimum-Disclosure Decompositions

As was mentioned above, some decompositions are better than others and some functions have efficient decompositions that are "as good as possible" (in that they do not leak anything beyond the ability to compute the residual functions g_i). Fix an n-input function f and n particular inputs $x_1, \ldots, x_n$, and recall that for all $i = 0, \ldots, n$ we denote by $g_i^{\vec{x}}$ the "residual function" with the first i variables fixed. That is, for $\vec{x} = \langle x_1, \ldots, x_n \rangle$, define

$$g_i^{\vec{x}}(z_{i+1}, \ldots, z_n) \stackrel{\text{def}}{=} f(x_1, \ldots, x_i, z_{i+1}, \ldots, z_n). \tag{3}$$

(In particular $g_0^{\vec{x}} = f$ and $g_n^{\vec{x}}$ is the constant function $g_n^{\vec{x}}(\cdot) = f(x_1, \ldots, x_n)$.) As we explained above, any decomposition of f must "at least leak the ability to compute $g_i^{\vec{x}}$" on all residual input vectors $z_{i+1}, \ldots, z_n$. A minimum-disclosure decomposition is one that does not leak anything else. Namely, for all i it is possible to compute the output of the composition of the first i functions $f_1, \ldots, f_i$, given only *oracle access* to the residual function $g_i^{\vec{x}}(\cdot)$.

Definition 2.2 (Minimum-Disclosure). *A decomposition $\bar{f}$ is* minimum disclosure *if there exists a probabilistic black box simulator S such that for every vector of inputs $\vec{x} = \langle x_1, \ldots, x_n \rangle$ of total length m and every $i \in [n]$,*

- *$S^{g_i^{\vec{x}}(\cdot)}(m, n, i)$ runs in time polynomial in $m + n$, and*
- *The output of $S^{g_i^{\vec{x}}(\cdot)}(m, n, i)$ equals $\tilde{f}_i(x_1, \ldots, x_i)$, except with negligible probability.*[1]

The notion of one composition which is better than another can be similarly defined via a simulator that can compute y_i' from y_i. This is omitted in the current version. We stress that not all functions have efficient minimum-disclosure decompositions,[2] as we now prove:

Theorem 2.3. *If one-way functions exist, then there are functions that do not have efficient minimum-disclosure decompositions.*

Proof. The proof follows from the observation that a decomposition is minimum-disclosure only when the residual functions g_i are efficiently learnable. We know from HILL [15] and GGM [13] that one-way functions imply pseudorandom functions, so consider a particular pseudorandom function $f : \mathsf{Seeds} \times \mathsf{Inputs} \to \mathsf{Outputs}$. We prove that the function f (when viewed as a two-input function $f(s, x)$, where the two inputs are the seeds and the input, respectively) does not have an efficient minimum-disclosure decomposition.

Assume to the contrary that $\langle f_1, f_2 \rangle$ is a minimum-disclosure decomposition. This means that (a) given $y_1 = f_1(s)$ is it possible to efficiently compute $f(s, x) = f_2(y_1, x)$ for any x, and (b) there is an efficient simulator S that can compute $y_1 = f_1(s)$ given oracle access to the function $f(s, \cdot)$.

But this immediately yields an attack on the pseudo-randomness of f. Given access to an oracle O, which computes either $f(s, \cdot)$ or a random function, perform the following procedure:

[1] In the case of randomized functionalities f, we require that $\{S^{g_i^{\vec{x}}(\cdot)}(m, n, i)\} \stackrel{c}{\equiv} \{\tilde{f}_i(x_1, \ldots, x_i)\}$.

[2] The residual truth table of a function is always minimum disclosure; however, it may be exponentially large.

First run the simulator S^O to get y_1, then choose some arbitrary input x^* that was not queried during the run of S^O and check whether $O(x^*) = f_2(y_1, x^*)$. If the oracle O implements $f(s, \cdot)$ then the answers must agree, whereas if O is a random function then with high probability they disagree. $\square$

Incomparable Decompositions. We also note that there are functions that seemingly do not have one best decomposition but rather many incomparable ones. For example, consider the Naor-Reingold pseudorandom function [19]. This is a construction over a group $\mathbb{G}$ of prime-order q with generator G where DDH is hard. The keys are $(n+1)$-vectors of indexes in $\mathbb{Z}_q$, and the function maps n-bit strings into the group $\mathbb{G}$.

$$f : \mathbb{Z}_q^{n+1} \times \{0,1\}^n \to \mathbb{G}, \qquad f(\langle a_0, a_1, \ldots, a_n \rangle, \langle \sigma_1, \ldots, \sigma_n \rangle) \stackrel{\text{def}}{=} G^{a_0 \cdot \prod_{i=1}^n a_i^{\sigma_i}}$$

Naor and Reingold proved that under DDH this is a pseudorandom function, and by the proof of Theorem 2.3 this means that it has no minimum-disclosure decomposition (when viewed as a two-input function). Moreover, it seem that "the only way" of computing this two-input function in a one-pass manner is for the first player to reveal all its indexes a_i. In fact, there are slightly less revealing decompositions: For every $i = 0, 1, \ldots, n$ there is a decomposition of f that reveals only G^{a_i} and not a_i itself (but still reveals all the other a_j's in the clear). Namely, we decompose f into $\bar{f}^{(i)} = \left\langle f_1^{(i)}, f_2^{(i)} \right\rangle$ such that $f_1^{(i)}(a_0, a_1, \ldots, a_n) = \langle a_0, \ldots, a_{i-1}, G^{a_i}, a_{i+1}, \ldots, a_n \rangle$. Clearly, given $f^{(i)}(\vec{a})$ (for any i) it is possible to compute the function $f(\vec{a}, \cdot)$, hence these are all valid decompositions of f. Also, it is easy to see that none of these values $f^{(i)}(\vec{a})$ can be computed from any other $f^{(i')}(\vec{a})$ if computing discrete-logarithms is hard in $\mathbb{G}$.

2.2 Examples of Functions with Minimum-Disclosure Decompositions

The sum function. Perhaps the simplest example is the sum function over a group: $f(x_1, \ldots, x_n) = \sum_{j=1}^n x_j$. In this case clearly the decomposition into partial sums $f_i(y_{i-1}, x_i) = y_{i-1} + x_i$ is minimum disclosure. Indeed, we have $\tilde{f}_i(x_1, \ldots, x_i) = \sum_{j=1}^i x_j$, and the simulator S can simply query $g_i^{\vec{x}}(0, \ldots, 0)$ and return the answer that it gets:

$$g_i^{\vec{x}}(0, \ldots, 0) \;=\; f(x_1, \ldots, x_i, 0, \ldots, 0) \;=\; \sum_{j=1}^{i} x_i \;=\; \tilde{f}_i(x_1, \ldots, x_i).$$

Selection functions. Other illustrating examples of functions with minimum-disclosure decompositions are the selection functions. Consider first the selection function with index at the end, $f(x_1, \ldots, x_{n-1}, j) = x_j$. Here we can see that the trivial decomposition, where for $i < n$ we have $f_i =$ identity and for $i = n$ we have $f_n = f$, is minimum disclosure. This is because given oracle access to $g_i^{\vec{x}}$ for any $i < n$, the simulator can just query it with varying inputs of the selection variable j, thus getting all the inputs $x_1, \ldots, x_i$.

On the other hand, consider the selection function with index at the beginning, $f(j, x_2, \ldots, x_n) = x_j$. Here a minimum disclosure decomposition would maintain a value and a state bit (wait/done), such that when the state is wait then the value is j, and when the state is done then the value is x_j. Namely, we have $f_1(j) = \langle j, \mathsf{wait} \rangle$, and for all $i > 1$

$$f_i(\langle \text{val}, \text{state} \rangle, x_i) = \begin{cases} \langle x_i,\ \mathsf{done} \rangle & \text{if state=}\mathsf{wait} \text{ and val} = i \\ \langle \text{val}, \text{state} \rangle & \text{otherwise} \end{cases}$$

(and of course the last function f_n omits the state bit). To see that this is indeed minimum disclosure, notice that given access to $g_i^{\vec{x}}$ the simulator can test if the selection index j is larger than i, e.g., by testing if $g_i^{\vec{x}}$ gives different values on $\langle 0, 0, \ldots, 0\rangle$ and $\langle 1, 1, \ldots, 1\rangle$. If $j > i$ then the simulator can find j by testing which is the input that $g_i^{\vec{x}}$ depends on, and if $j < i$ the the simulator can output x_j (which is the output of $g_i^{\vec{x}}$ on every input).

The general case of a selection function with index in the middle, $f(x_1, \ldots, x_{t-1}, j, x_{t+1}, \ldots, x_n) = x_j$, can be obtained from the two previous cases. A minimum-disclosure decomposition will have f_i=identity for $i < t$, $f_t(x_1, \ldots, x_{t-1}, j)$ computes the appropriate $\langle$value,state$\rangle$ pair, and the rest of the f_i's defined as in the case of index at the beginning.

Binary symmetric functions. An n-input binary symmetric function takes n bits as input, and the output depends only on the number of 1's in the input (i.e., the Hamming weight). Some examples include the AND, OR, PARITY, and MAJORITY functions. We note that the truth table of a binary symmetric function has an efficient representation: we just list for every $0 \leq j \leq n$ the output of f on inputs with Hamming-weight n. Thus, the truth table is of length $n + 1$ rather than of length 2^n. We also note that for a binary symmetric function f and input $\vec{x}$, all the corresponding $g_i^{\vec{x}}$'s are also binary symmetric functions, and moreover the truth table of $g_{i+1}^{\vec{x}}$ can be computed from the value of x_i and the truth table of $g_i^{\vec{x}}$. Specifically, for $x_i = 0$ the truth table of $g_{i+1}^{\vec{x}}$ is obtained from that of $g_i^{\vec{x}}$ by removing the last row, and for $x_i = 1$ the truth table of $g_{i+1}^{\vec{x}}$ is obtained by removing the first row from that of $g_i^{\vec{x}}$.

For a binary symmetric function f, consider the decomposition that outputs at every step i the truth table of $g_i^{\vec{x}}$. The above observations implies that this decomposition is efficient, and it is minimum disclosure since it is easy to compute the truth table of a symmetric function given oracle access to that function.

To illustrate this concretely, consider the MAJORITY function over 3 inputs. The truth table (in vector form) equals $(0, 0, 1, 1)$ where the ith entry corresponds to inputs of Hamming weight i. Now, $g_1^{(0,\cdot,\cdot))}(z_2, z_3) = (0, 0, 1)$ and $g_1^{(1,\cdot,\cdot)}(z_2, z_3) = (0, 1, 1)$; $g_2^{(0,0,\cdot))}(z_3) = (0, 0)$, $g_2^{(0,1,\cdot))}(z_3) = (0, 1)$, $g_2^{(1,0,\cdot))}(z_3) = (0, 1)$, and $g_2^{(1,1,\cdot))}(z_3) = (1, 1)$. As can be seen, the truth table in each step is obtained by removing the first or last element. In order to carry out the above for the PARITY function, just do the same when starting with truth table $(0, 1, 0, 1)$.

Symmetric functions over other domains. The observations from above can be extended to symmetric functions over other domains. We assume without loss of generality that the domain is $\mathbb{Z}_c = \{0, 1, \ldots, c - 1\}$ for some integer c. An n-input symmetric function over $\mathbb{Z}_c$ is one where permuting the inputs does not affect the output. In other words, the output depends only on how many of the inputs assume what value of the domain. This type of function is common for statistical measurement, including functions like SUM, AVERAGE, MEDIAN, MAJORITY, MAXIMUM and more.

The truth table for a symmetric function over $\mathbb{Z}_c$ can be expressed using a single row for all the inputs that have exactly j_0 inputs of value 0, j_1 inputs of value 1, and so on up to j_{c-2} inputs of value $c - 2$ and $j_{c-1} = n - \sum_{i=0}^{c-2} j_i$ inputs of value $c - 1$. That is, we have a row in the truth table for every c-vector of non-negative integers $\langle j_0, j_1, \ldots, j_{c-1}\rangle$ that sum up to n, so we have a total of $\binom{n+c-1}{n}$ rows. Hence the truth table is of polynomial-size $O(n^c)$ for any constant c. Moreover, in this case we again have the properties that all the $g_i^{\vec{x}}$'s are symmetric, and the truth table of $g_{i+1}^{\vec{x}}$

can be computed efficiently from the value of x_i and the truth table of $g^{\vec{x}}_{i+1}$ (see Section 4.2 for more details).

Also similarly to the binary case, when the truth table has polynomial size then it can be constructed efficiently given only oracle access to the function, hence the functions that output at every step i the truth table of $g^{\vec{x}}_i$ constitute a minimum-disclosure decomposition of the original symmetric function f.

3 Server-Based One-Pass Protocols

All our protocols are staged in the PKI model. We assume that in an initial setup phase each honest party has chosen a public and a private key according to a known key-generation algorithm, and the public key is made known to all the other parties. After seeing the public keys of all the honest parties, the dishonest parties get to choose their own public keys (in any way that they see fit) and these public keys too are made known to everyone. Hence, at the onset of the protocol each party knows the public keys of all other parties, and each honest party knows the private key corresponding to its own public key.

A server-based one-pass protocol for n clients and a server is in fact a sequence of n two-party protocols, $\bar{\pi} = \langle \pi_1, \ldots, \pi_n \rangle$, which are carried out sequentially with π_i being a two-party protocol between the server and the ith client P_i. In our model the input to the ith honest client consist of its private key and the public keys of all the other parties, and also another input x_i which is its input to the function f. The input of the server is its private key and the public keys of all the clients, and also its output from the previous protocol π_{i-1} (which is empty for π_1). We model the setting where the server has no input to f; if it does have input, then it can play both client P_n and the server. The output of the protocol $\bar{\pi}$ is defined as the output of the server after the last protocol π_n. Below we denote the clients by $P_1, P_2, \ldots, P_n$ and the server by P_{n+1}. We denote the joint outputs of an adversary $\mathcal{A}$ and server P_{n+1} after a real execution of $\bar{\pi}$ with inputs $\vec{x} = (x_1, \ldots, x_n)$, vector of public/private key-pairs $\vec{kp}$, auxiliary input z to $\mathcal{A}$, corrupted parties $I \subseteq [n+1]$, and security parameter m, by $\text{REAL}_{\bar{\pi},\mathcal{A}(z),I}(\vec{x}, \vec{kp}, 1^m)$.

Securely computing a decomposition. We define security via the ideal/real paradigm in the stand-alone setting with static corruptions; the extension to composition and adaptive corruptions is left for future work. In the ideal world, there is an additional trusted party that carries out the computation for the parties. In our setting, the trusted party receives the input from all the clients, and the identities of the corrupted parties, and sends the function output to the server together with any additional information that is inherently learned in our model (based on who is corrupted). We stress that the ideal model is defined for a *function decomposition* $\bar{f}$. (It is not necessary to include f since $\bar{f}$ fully determines f.)

In the ideal world of the *semi-honest model*, the output that is given to the server is always the value of the function $f(x_1, \ldots, x_n)$ on the given inputs of all the clients. (Since the parties are semi-honest, the inputs used by the clients in the protocol equal those that appear on their input tapes.) In addition, if the server is corrupted, then the trusted party sends it the value $\tilde{f}_i(x_1, \ldots, x_i) = f_i(\cdots, f_2(f_1(x_1), x_2) \cdots, x_i)$ where i is the index of the last honest party. We denote the outputs of a semi-honest ideal-world adversary $\mathcal{S}$ and server P_{n+1} after an ideal execution with inputs $\vec{x} = (x_1, \ldots, x_n)$, auxiliary input z to $\mathcal{S}$, corrupted parties $I \subseteq [n+1]$, and security parameter m, by $\text{IDEAL}^{\text{SH}}_{\bar{f},\mathcal{S}(z),I}(\vec{x}, z, 1^m)$.

The ideal-world of the malicious model is exactly the same, except that corrupted clients may send any arbitrary inputs to the trusted party, not necessarily the ones from their input. By convention, if a client sends input $\perp$, then the output of the function is defined to be $\perp$ (representing an aborted execution). The joint output here is denoted $\text{IDEAL}^{\text{MAL}}_{\bar{f},\mathcal{S}(z),I}(\vec{x}, z, 1^m)$.

Definition 3.1 (Securely Computing a Decomposition)**.** *Let f be an n-input function and let $\bar{f} = \langle f_1, \ldots, f_n \rangle$ be a decomposition of f. A server-based one-pass protocol $\bar{\pi}$* securely computes the decomposition *$\bar{f}$ in the semi-honest (resp. malicious) model, if for every non-uniform probabilistic polynomial-time semi-honest (resp. malicious) adversary $\mathcal{A}$ in the real world, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ for the semi-honest (resp. malicious) ideal world, such that for all $\vec{x} \in (\{0,1\}^*)^n$ and $z \in \{0,1\}^*$*

$$\left\{\text{IDEAL}_{\bar{f},\mathcal{S}(z),I}(\vec{x}, 1^m)\right\} \stackrel{c}{\equiv} \left\{\text{REAL}_{\bar{\pi},\mathcal{A}(z),I}(\vec{x}, \vec{kp}, 1^m)\right\}$$

where the key-pairs $\vec{kp}$ are chosen as described above.

We stress that if the server is honest, then in all cases nothing is learned by the adversary. We also note that a protocol that securely computes a decomposition $\bar{f}$ is only as good as its decomposition. When the function has a minimum-disclosure decomposition and we have a protocol that realize it, then we say call this protocol an optimally-private protocol.

Definition 3.2 (Optimally-Private)**.** *Let f be an n-input function. We say that $\bar{\pi}$ is an* optimally-private server-based one-pass protocol for computing *f if there exists a minimum-disclosure decomposition $\bar{f}$ of f such that $\bar{\pi}$ securely computes $\bar{f}$ in the semi-honest (resp. malicious) model.*

We remark that for the case of *optimal* server-based one-pass protocols, an equivalent way of defining the ideal model is that in the case that the server is corrupted and P_i is the last honest party, it can query the trusted party with $z_{i+1}, \ldots, z_n$ and receive back $g_i^{\vec{x}}(z_{i+1}, \ldots, z_n)$ as many times as it wishes (where $\vec{x}$ are the inputs actually sent to the trusted party in the ideal execution).

4 Practical Optimal Protocols

In Section 5 below, we show that *any* decomposition can be securely computed given a public-key infrastructure, under the DDH assumption. In particular, any function that has a minimum-disclosure decomposition can be computed with optimal privacy. However, this construction is far from being practical; even for simple functions and semi-honest adversaries, it requires computing hundreds of exponentiations per gate. In this section, we present highly efficient protocols for specific examples from Section 2.2. These protocol are truly practical and could be implemented, for example, in a conference program committee review site in order to carry out secure voting. (With only a few tens of users, the solution that provides security in the presence of malicious adversaries would only require a few seconds of computation by each client and the server.)

4.1 Protocols for Symmetric Functions

We begin by showing how to securely compute any binary symmetric function, based on the truth-table decomposition described in Section 2.2.

4.1.1 The Semi-Honest Case

Recall that symmetric functions have a concise truth table of size $n + 1$, and that the minimum-disclosure decomposition for functions of this class consists of the truth table of the $g_i^{\vec{x}}$'s, and that computing the next truth table is done by removing the first or last row of the current truth table. Intuitively, our protocol works by having the first client P_1 encrypt each entry of the truth table iteratively (in an onion like structure) under all parties' public keys. Then, each party in turn removes the encryption under its public key, and removes the first row of the truth table if its input is 0 and last row of the truth table if its input is 1. After the last player, the table contains just one row which is encrypted under the server's key, so the server can decrypt it and output the right value. (We remark that if all parties should receive output, then the server can just send the output to them.)

Note that if the server and last $n - i$ parties are corrupted, then they can decrypt the truth table that they receive; however, this is exactly $\tilde{f}(x_1, \ldots, x_i)$ which is what they are allowed to receive. This solution is not quite enough, however. For example, a collusion of P_1 and P_3 can learn P_2's exact input (irrespective of whether or not the server is corrupted). To see this, observe that P_1 generates all the ciphertexts, so in particular it can see all the P_3 ciphertexts, as they will be seen by P_3 after P_2 decrypts its layer of encryption. Hence, given P_3's view P_1 can determine if P_2 removed the first or the last row of the table.

We solve this problem by using rerandomizable public-key encryption; loosely speaking, this means that given an encryption $c = E_{pk}(x)$ and the public key pk it is possible to generate an equivalent encryption $c' = E_{pk}(x)$ with *independent* randomness. We stress that the rerandomization must work on all layers of the (onion-type) encryption. The requirements here are therefore different from the standard notion. Let $E_{pk}(x; r)$ denote an encryption of x using randomness r, and let $E_{pk_1,\ldots,pk_{n+1}}(x; r_1, \ldots, r_{n+1}) = E_{pk_1}(\cdots E_{pk_{n+1}}(x; r_{n+1}) \cdots ; r_1)$ denote a layered encryption starting with the encryption of x under pk_{n+1} with randomness r_{n+1} and re-encrypting under each pk_i in turn, using randomness r_i. For shorthand, we write $\bar{E}_{\vec{pk}}(x; \vec{r})$ where $\vec{pk} = (pk_1, \ldots, pk_{n+1})$ and $\vec{r} = (r_1, \ldots, r_{n+1})$. [3] We define:

Definition 4.1. *A public-key scheme* (G, E, D) *is* layer rerandomizable *if there exists a procedure* $\mathcal{R}$ *such that for every* $x \in \{0, 1\}^*$ *and every* $\vec{r} \in (\{0, 1\}^*)^n$,

$$\left\{\vec{pk}\ , \bar{E}_{\vec{pk}}(x; \vec{r}),\ \bar{E}_{\vec{pk}}(x; \vec{r}')\right\} \quad \equiv \quad \left\{\vec{pk},\ \bar{E}_{\vec{pk}}(x; \vec{r}),\ \mathcal{R}(\vec{pk}, \bar{E}_{\vec{pk}}(x; \vec{r}))\right\}$$

where $\vec{pk} = (pk_1, \ldots, pk_n)$ *is such that all the* pk_i*'s are in the range of* G*, and* $r \in_R (\{0, 1\}^*)^n$ *is a vector of uniformly distributed random strings.*

We stress that the definition requires the rerandomization to work for *all* randomness $\vec{r}$ (even randomness that is "badly chosen"). However, it is assumed that all the public keys are "legitimate" in that they are in the range of G. In the case of malicious adversaries, a proof may need to be added that the keys are legitimate in order to ensure that the rerandomization works.

Layer rerandomizability can be obtained from any additively homomorphic encryption scheme. Namely, define an initial layered encryption of x by

$$\bar{E}_{\vec{pk}}(x; \vec{r}) \ \stackrel{\text{def}}{=}\ \langle E_{pk_1}(x_1; r_1), \ldots, E_{pk_n}(x_n; r_n)\rangle$$

[3] Below we abuse these notations somewhat, denoting by $\bar{E}_{\vec{pk}}(x; \vec{r})$ a procedure that encrypts x under all the public keys but not necessarily in an onion fashion.

where $x_1, \ldots, x_n$ are chosen at random under the constraint that $\oplus_{j=1}^{n} x_j = x$. A *j'th step layered encryption* of x is defined as

$$\bar{E}^j_{\vec{pk}}(x; \vec{r}) \stackrel{\text{def}}{=} \left\langle x_1, \ldots, x_j, E_{pk_{j+1}}(x_{j+1}; r_{j+1}), \ldots, E_{pk_n}(x_n; r_n) \right\rangle$$

We will typically refer generally to layered encryption, assuming that jth level layered encryption is used after the interaction of p_j with the server. Rerandomization works by adding to the x_i's random δ_i's that sum up to zero, and then rerandomizing each ciphertext separately, under the appropriate key. In addition, it is possible to decrypt in layers by having each party decrypt its ciphertext in turn and pass on the decrypted value along with the rest. Namely, the jth party transform a $(j-1)$th level layered encryption to a jth level layered encryption. Below we show that using El Gamal it is possible to work more efficiently than this.

An optimally-private protocol for binary symmetric functions and semi-honest adversaries, using layer rerandomizable encryption, appears in Protocol 4.2.

PROTOCOL 4.2. (Semi-Honest Optimal Protocol for a Binary Symmetric f)

- **Inputs:** Each party P_i ($1 \leq i \leq n$) has a private input $x_i \in \{0, 1\}$, its own private key sk_i, and a vector of public keys $(pk_1, \ldots, pk_{n+1})$; the server P_{n+1} has sk_{n+1} and $(pk_1, \ldots, pk_{n+1})$.
- **The protocol:**
  1. *Truth table initialization – interaction of P_1 with P_{n+1}:*
     (a) Party P_1 constructs the truth table $\mathcal{T} = (t_0, \ldots, t_n)$ where $t_i = f(1^i 0^{n-i})$.
     (b) Party P_1 removes the last element of $\mathcal{T}$ if $x_1 = 0$ and the first element of $\mathcal{T}$ if $x_1 = 1$. Denote the result by $\mathcal{T}_1 = (t'_1, \ldots, t'_n)$.
     (c) For every $j = 1, \ldots, n$, P_1 computes $c_j = \bar{E}_{\vec{pk}}(t'_j)$ and sends the encrypted truth table $\mathcal{C}_1 = (c_1, \ldots, c_n)$ to the server P_{n+1}.
  2. *Interaction of clients $P_2, \ldots, P_n$ with server P_{n+1}:* For $i = 2, \ldots, n$, party P_i interacts with the server P_{n+1} as follows:
     (a) P_{n+1} sends P_i the encrypted truth table $\mathcal{C}_{i-1}$ of length $n - i + 1$.
     (b) P_i removes the last element of $\mathcal{C}_{i-1}$ if $x_i = 0$ or the first element if $x_i = 1$.
     (c) P_i decrypts a layer of all of the remaining ciphertexts in $\mathcal{C}_{i-1}$ using its secret key sk_i; denote the result by $\mathcal{C}'_i$.
     (d) P_i rerandomizes all of the ciphertexts in $\mathcal{C}'_i$ using the public key pk_{i+1}; denote the result by $\mathcal{C}_i$.
     (e) P_i sends $\mathcal{C}_i$ back to the server.
  3. *Concluding the computation:* Upon receiving the encrypted truth table $\mathcal{C}_n$ of length 1 from P_n, the server P_{n+1} decrypts the ciphertext using its secret key sk_{n+1} and outputs the result.

Theorem 4.3. *Let f be a binary symmetric function. If the encryption scheme (G, E, D) is layer rerandomizable, and all honest parties' public keys are generated honestly using G, then Protocol 4.2 is an optimally-private server-based one-pass protocol for computing f, in the presence of semi-honest adversaries. Moreover, it is secure even if the semi-honest adversary can choose the randomness for the protocol in an arbitrary manner.*

Proof (sketch). We separately prove the case that P_{n+1} is corrupted and the case that it is not. If P_{n+1} is not corrupted, then it suffices to prove that it obtains correct output and that the adversary's view can be simulated without any help from the trusted party. Correctness is immediate from the construction. The view of the adversary can be simulated since everything is encrypted under the key of the honest server. Specifically, every time an honest party P_i is supposed to carry out its interaction with the server, construct a brand new truth table $\mathcal{C}_i$ which contains $n-i+1$ encryptions of 0 under the public-keys $pk_{i+1}, \ldots, pk_{n+1}$, in turn. The fact that this is indistinguishable from a real execution follows directly from the hiding property of encryptions, and the rerandomizability property.

Next, we consider the case that the server P_{n+1} is corrupted, and $1 \leq i \leq n$ is the index of the last honest party. In this case, the simulator $\mathcal{S}$ is given the value $y_i = \tilde{f}_i(x_1, \ldots, x_n)$, which in this case is the appropriate partial truth table. The simulation is the same as before for every iteration up to and including $i-1$. In the ith iteration, $\mathcal{S}$ simulates the message sent by the honest P_i by encrypting under the public keys $pk_{i+1}, \ldots, pk_{n+1}$ the partial truth table that it received from the trusted party. As before, the output distribution of the adversary is indistinguishable from a real execution (note that the last simulated message is actually identical to in a real execution; the difference comes from prior ones which are all encryptions of 0 instead of the real partial truth table). □

A concrete instantiation. A simple and highly efficient layer rerandomizable encryption scheme can be constructed from El Gamal. Let $\mathbb{G}$ be a group of prime order q with generator G. Then, for public-key $h = G^\alpha$ and $E_{pk}(x) = (G^r, h^r \cdot x)$, define $\mathcal{R}(pk, \langle u, v \rangle) = \langle u \cdot G^s, v \cdot h^s \rangle$, where $s \in_R \mathbb{Z}_q$. Observe that for $u = G^r, v = h^r \cdot x$ it follows that $\mathcal{R}(pk, u, v) = (G^{r+s}, h^{r+s} \cdot x)$, which is distributed identically to an encryption of x under an independent random string $r' = r + s \bmod q$.

In order to make this layer rerandomizable without increasing the size of the ciphertext, we define layered encryption as follows. Each party P_i has an El Gamal public-key $h_i = G^{\alpha_i}$ relative to the same group $(\mathbb{G}, q, g)$ as before. However, an encryption of x under the public keys $h_1, \ldots, h_n$ is defined to be $(G^r, (H_{1,n})^r \cdot x)$, where $H_{1,n} = \prod_{j=1}^n h_j = G^{\sum_{j=1}^n \alpha_j}$. In general, we define $H_{i,n} = \prod_{i=1}^n h_i = G^{\sum_{j=i}^n \alpha_j}$. It remains to show how P_i "decrypts" under its key h_i and rerandomizes the result. Given (u, v) where $u = G^r$ and $v = (H_{i,n})^r \cdot x$, party P_i decrypts by computing

$$u' = u \qquad \text{and} \qquad v' = v \cdot u^{-\alpha_i}.$$

This works because taking $u = G^r$ and $v = x \cdot (H_{i,n})^r$ we have that

$$v \cdot u^{-\alpha_i} = x \cdot (H_{i,n})^r \cdot (G^r)^{-\alpha_i} = x \cdot \left(G^{\sum_{j=i}^n \alpha_j}\right)^r \cdot \left(G^{-\alpha_i}\right)^r = x \cdot \left(G^{\sum_{j=i+1}^n \alpha_j}\right)^r = x \cdot (H_{i+1,n})^r$$

and so (u', v') is a valid encryption of x with randomness r, under public key $H_{i+1,n}$. Rerandomization is then carried out as described above, using public-key $H_{i+1,n}$. That is, we compute $u'' = u' \cdot G^s$ and $v'' = v' \cdot (H_{i+1,n})^s$.

Efficiency. Party P_1 carries out just n encryptions since it first computes $H_{1,n}$ at the cost of n multiplications (which is not significant), and then computes each ciphertext at the cost of 2 exponentiations. Next, each P_i computes $n-i$ decryptions and rerandomizations at a total cost

of $3(n-i)$ exponentiations.[4] Finally, the overall communication complexity is $O(n^2)$ ciphertexts which equals $n(n-1)$ group elements, but each party P_i only receives $n-i+2$ ciphertexts and returns $n-i+1$ ciphertexts. This can be carried out in practice, even for n's in the tens of thousands. (Using El Gamal over an Elliptic curve group, each encryption/decryption costs just a few milliseconds.)

Order of parties. Observe that using our concrete instantiation, the parties can connect and interact with the server in any order. This is an important property for practical implementation and deployment.

4.1.2 The Malicious Case

There are a number of possible attacks in the malicious case. First, P_1 can generate an incorrect truth table and break correctness. Likewise, any intermediate corrupted P_i can just send $\mathcal{C}_{i+1}$ that is generated from scratch. We solve these problems by having the parties prove that they have generated everything correctly. This requires the use of zero-knowledge proofs of knowledge, which are reminiscent of those needed in mix-net type constructions. However, in our setting, these proofs must be non-interactive so that an intermediate party P_i can verify that *all* of the actions of parties $P_1, \ldots, P_{i-1}$ were carried out correctly. We first describe the protocol in general terms and prove its security, and then show how to efficiently instantiate all the components. Unlike the generic construction with malicious security of Section 5.4, here we use a random oracle in order to obtain efficient non-interactive zero-knowledge proofs of knowledge via the Fiat-Shamir paradigm. We will also need each party to have a pair of keys for a secure digital signature scheme, to make sure that no intermediate party reconstructs everything from scratch, effectively changing the inputs of prior parties.

Although this strategy sounds like it must be computationally very expensive, we will show that it can be carried in reasonable time, and that it can be practical for n's that are not too large. Full details appear in Protocol 4.4. We note that we separate the interaction of P_1 into distinct steps for the sake of clarity only; the messages can be sent together.

[4] If each party P_i defines $i-1$ public keys, then encryption can be modified to use a single G^r value for all encryptions, where all H_i values are raised to the same power of r. This, in turn, reduces the overhead of each P_i to $2(n-i)$ exponentiations.

PROTOCOL 4.4. (Malicious Optimal Protocol for a Binary Symmetric Function f)

- **Inputs:** Each party P_i ($1 \leq i \leq n$) has a private input $x_i \in \{0,1\}$, its own private keys sk_i and sk_i' for encryption and signing, and a vector of public keys $(pk_1, pk_1', \ldots, pk_n, pk_n', pk_{n+1})$; the server P_{n+1} has one secret key sk_{n+1} and $(pk_1, pk_1', \ldots, pk_n, pk_n')$. In addition, all parties have a common reference string for a system of non-interactive zero-knowledge proofs of knowledge.

  We assume that all of the public keys are certified, meaning that it is guaranteed that they are all legitimately constructed, and all parties verify all keys (for El Gamal this simply involves verifying that the public-key is in the group).

- **The protocol:**

  1. *Truth table initialization – interaction of P_1 with P_{n+1}:*
     (a) Party P_1 constructs the truth table $\mathcal{T} = (t_0, \ldots, t_n)$ where $t_i = f(1^i 0^{n-i})$.
     (b) For every $j = 0, \ldots, n$, P_1 computes $c_j = \bar{E}_{\vec{pk}}(t_j')$ and sends the encrypted truth table $\mathcal{C}_1 = (c_0, \ldots, c_n)$ to the server P_{n+1}.
     (c) Party P_1 generates a non-interactive zero-knowledge proof of knowledge that $\mathcal{C}_0$ was generated correctly; denote the proof string by ρ_0. P_1 signs on the pair $(\mathcal{C}_0, \rho_0)$ using its digital signing key sk_1'; denote the signature by σ_0.
     (d) Party P_1 sends $(\mathcal{C}_0, \rho_0, \sigma_0)$ to the server P_{n+1}.
     (e) P_{n+1} verifies the signature and proof; if they are not correct it halts and outputs $\perp$.
  2. *Interaction of clients $P_1, \ldots, P_n$ with server P_{n+1}:* For $i = 1, \ldots, n$, party P_i interacts with the server P_{n+1} as follows:
     (a) P_{n+1} sends P_i the tuples $(\mathcal{C}_0, \rho_0, \sigma_0), \ldots, (\mathcal{C}_{i-1}, \rho_{i-1}, \sigma_{i-1})$.
     (b) P_i verifies the proof and signature for all $j = 0, \ldots, i-1$. If they are not all correct, it halts and outputs $\perp$.
     (c) P_i removes the last element of $\mathcal{C}_{i-1}$ if $x_i = 0$, or the first element of $\mathcal{C}_{i-1}$ if $x_i = 1$.
     (d) P_i decrypts all of the remaining ciphertexts in $\mathcal{C}_{i-1}$ using its secret key sk_i; denote the result by $\mathcal{C}_i'$.
     (e) P_i rerandomizes all of the ciphertexts in $\mathcal{C}_i'$ using the public key pk_{i+1}; denote the result by $\mathcal{C}_i$.
     (f) P_i generates a proof ρ_i that $\mathcal{C}_i$ is a rerandomization of $\mathcal{C}_{i-1}$ minus either the first or last row. P_i generates a signature σ_i on $(\mathcal{C}_i, \rho_i)$ using its signing key sk_i'.
     (g) P_i sends $(\mathcal{C}_i, \rho_i, \sigma_i)$ back to the server.
     (h) P_{n+1} verifies the proof ρ_i and signature σ_i; if they are not both correct, it halts and outputs $\perp$.
  3. *Concluding the computation:* The server P_{n+1} decrypts the remaining ciphertext in $\mathcal{C}_n$ using its secret key sk_{n+1} and outputs the result.

We remark that if all parties should receive output, then P_{n+1} can send all of the tuples $(\mathcal{C}_1, \rho_1, \sigma_1), \ldots, (\mathcal{C}_n, \rho_n, \sigma_n)$ along with the output value y and a proof ρ_{n+1} that the remaining ciphertext in $\mathcal{C}_n$ is an encryption of the value y. As in the semi-honest case, the concrete instantiation of the encryption scheme must be specified; we use the same variant of El Gamal here.

Theorem 4.5. *Let f be a binary symmetric function. Assume that the encryption scheme (G, E, D) is layer rerandomizable, the digital signature scheme is existentially unforgeable, that all honest*

parties' public keys are generated honestly using G, and the proof system is a non-interactive zero-knowledge proof of knowledge. Then, Protocol 4.4 is an optimally-private server-based one-pass protocol for computing f, in the presence of malicious adversaries.

Proof (sketch). Since the semi-honest version of this protocol is secure for every chose of the randomness by the corrupted players, then adding noninteractive zero-knowledge proofs of correct behavior is sufficient to get security in the malicious model. □

Efficiency. The complexity of this protocol is significantly higher than for the semi-honest case. However, if the proofs are efficient then this protocol, too, is practical for n's in the hundreds. Assume for now that proving and verifying a table $\mathcal{C}_i$ of length $n - i$ requires $O(n - i)$ group exponentiations. We have that each party carries out work that is quadratic in n (in contrast to work that is linear in n as in the semi-honest case). Specifically, the ith party ($1 \leq i \leq n$) carries out i proof and signature verifications, and generates one proof of length $n - i$. Since the verification of a table of length $n - i$ requires $O(n - i)$ exponentiations, we have that party P_i computes $\sum_{j=1}^{i} O(n - j)$ exponentiations. Thus, party P_n computes $O(n^2)$ exponentiations (overall the complexity is cubic but since each party works completely separately in our setting, the individual work is more significant). In addition, the communication complexity is the transmission of up to $O(n^2)$ group elements. We use the following tools:

- *El Gamal proofs:* This include proving that an El Gamal ciphertext (u, v) is an encryption of a given value z under public key h, proving that (u, v) is decrypted as z under the secret key corresponding to public key h, proving that two ciphertexts encrypt the same value under h, etc. It is known that all these proofs can be converted into a proof that some tuple of the form (G, h, a, b) is a Diffie-Hellman tuple. Applying the Fiat-Shamir transform to the known Sigma protocol for this language, we have that constructing the proof requires 2 exponentiations, verifying the proof requires 4 exponentiations, and the length of the proof is 2 group elements and one element of $\mathbb{Z}_q$.
- *Proofs of compound statements:* The AND of L statements costs L times an individual statement (by just repeating and using the same query for each), and the OR of 2 statements costs about twice an individual statement using the method of Cramer et al. [5].

For details on how to carry out the above and their exact cost, see [6] or [16, Chapter 6].

We begin by showing how P_1 can prove the construction of the encrypted truth table. This is just the AND of n statements of the type "this ciphertext is an encryption of the value $f(1^\ell 0^{n-\ell})$ under the public-key $H_{1,n}$". This proof therefore costs $2n$ exponentiations to generate and $4n$ exponentiations to verify.

Next, we show how P_i can prove that it computed $\mathcal{C}_i$ correctly from $\mathcal{C}_{i-1}$. We first show how to prove that a single ciphertext is correctly computed. Recall that the "decryption" step of P_i is the computation $v' = v \cdot u^{-\alpha_i}$. Next, P_i rerandomizes (u, v') into (u'', v''). By what we have described above, these two steps involve proofs that $(u, v \cdot u^{\alpha_i}, u'', v'')$ is a Diffie-Hellman tuple. Thus, this is a conjunction of two Diffie-Hellman proofs at the overall cost of 4 exponentiations to prove and 8 exponentiations to verify. Now, observe that in order to prove that $\mathcal{C}_i$ is correctly computed from $\mathcal{C}_{i-1}$, party P_i must prove that it is either "decrypted and rerandomized" from the first $n - i +$ elements of $\mathcal{C}_i$ or the last $n - i + 1$ elements of $\mathcal{C}_i$. Thus, this is an OR of $3(n - i + 1)$ conjunctions.

The overall cost is $2(n-i+1)$ Diffie-Hellman proofs, or $4(n-i+1)$ exponentiations to prove and $8(n-i+1)$ exponentiations to verify.

We conclude that party P_1 needs to compute $2n+4(n+1)$ exponentiations (to prove the truth table and that it removed the first or last row), each P_i needs to compute less than $4n+8in$ exponentiations in order to verify what it received and an additional $4(n-i+1)$ exponentiations to prove its step. We therefore have an upper bound of $8n^2$ exponentiations per party, achieving our goal. We stress that the constant here is low, making this truly practical for n not too large. For example, with $n=100$, we have at most $8 \times 100^2 = 80,000$ exponentiations which can be computed in about 2.5 minutes.

Order of parties. As for the semi-honest case, using our concrete instantiation the parties can connect and interact with the server in any order.

4.2 Symmetric Functions over $\mathbb{Z}_c$

In this section, we show how to extend the techniques used for *binary* symmetric functions to those over $\mathbb{Z}_c$ where c is a constant. The first important observation is that a symmetric function of n variables in the range $\{0,\ldots,c-1\}$ has a truth table of size $\binom{n+c-1}{n}$, where each entry corresponds to a vector of c values in the range $\{0,\ldots,n\}$, such that the sum of all entries is n. This is because the value of $f(x_1,\ldots,x_n)$ depends only on how many times each of the values $0,\ldots,c-1$ appears among $x_1,\ldots,x_n$. Thus, each input vector $x_1,\ldots,x_n$ is mapped to the vector $(a_1,\ldots,a_c)$ such that $i \in \mathbb{Z}_c$ appears a_i times in the vector $x_1,\ldots,x_n$. The number of different vectors that can be constructed in this way is $\binom{n+c-1}{n}$.

The protocol for the semi-honest case is identical to protocol 4.1.1, with the following modifications: P_1 constructs a truth table according to the concise encoding of the table described above. Then, in Steps 1(b) and 2(b), instead of removing the first or last entries of the table, the parties remove a larger subset of the entries, as we describe next.

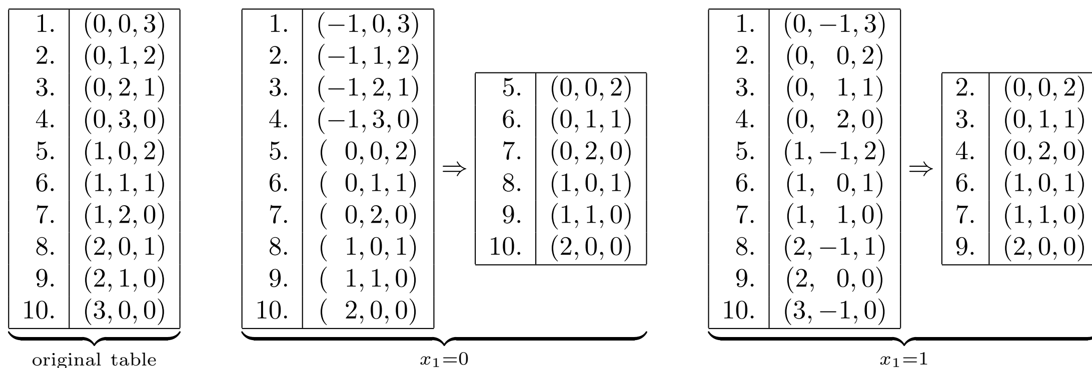
The truth table. We arrange the truth table of $\binom{n+c-1}{n}$ rows in lexicographic order. For example, for $n=c=3$ we would have the following table:

Line #	frequency vector	function value
1.	$(0,0,3)$	$f(2,2,2)$
2.	$(0,1,2)$	$f(1,2,2)$
3.	$(0,2,1)$	$f(1,1,2)$
4.	$(0,3,0)$	$f(1,1,1)$
5.	$(1,0,2)$	$f(0,2,2)$
6.	$(1,1,1)$	$f(0,1,2)$
7.	$(1,2,0)$	$f(0,1,1)$
8.	$(2,0,1)$	$f(0,0,2)$
9.	$(2,1,0)$	$f(0,0,1)$
10.	$(3,0,0)$	$f(0,0,0)$

Each entry in the frequency-vector column is of the form (a_0,a_1,a_2), where a_i encodes the number of times that the character i appears, and $a_0+a_1+a_2=n=3$. (Note that frequency-vector column contains only the indexes of the entries. The entry itself includes the output of the function for this specific input.)

P_1 must send to P_2 the reduced truth table where the entries that are inconsistent with P_1's input are removed. That table encodes exactly the truth table of the residual function $f(x_i, \cdot, \cdot)$. Clearly, the reduced table also encodes a symmetric function over $\mathbb{Z}_c = \mathbb{Z}_3$, this time with only $n-1 = 2$ inputs. Moreover, when P_1 removes the inconsistent entries we get the truth table of the residual function *again ordered by lexicographic order*, no matter what was P_1's input. Hence P_2 can continue by again removing the entries that are inconsistent, with his input without having to know which rows were removed by P_1.

One way to see this visually is to imagine the following procedure that P_1 employs for reducing the truth table: on input $x_1 = i \in \mathbb{Z}_c$, P_1 first subtracts one from the i'th coordinate in each entry of the frequency-vector column (to account for the fact that the input $x_i = i$ will be hard-wired in the residual function and should not be counted anymore), and then removes the rows corresponding to entries with negative numbers. (These are the inconsistent rows: since P_1 has input i then consistent frequency-vectors couldn't have zero in the i'th position.) An illustration of this transformation for the cases where P_1's input is $x_1 = 0$ and $x_1 = 1$ can be found below:



original table

1.	$(0,0,3)$
2.	$(0,1,2)$
3.	$(0,2,1)$
4.	$(0,3,0)$
5.	$(1,0,2)$
6.	$(1,1,1)$
7.	$(1,2,0)$
8.	$(2,0,1)$
9.	$(2,1,0)$
10.	$(3,0,0)$

$x_1=0$

1.	$(-1,0,3)$
2.	$(-1,1,2)$
3.	$(-1,2,1)$
4.	$(-1,3,0)$
5.	$(\ 0,0,2)$
6.	$(\ 0,1,1)$
7.	$(\ 0,2,0)$
8.	$(\ 1,0,1)$
9.	$(\ 1,1,0)$
10.	$(\ 2,0,0)$

$\Rightarrow$

5.	$(0,0,2)$
6.	$(0,1,1)$
7.	$(0,2,0)$
8.	$(1,0,1)$
9.	$(1,1,0)$
10.	$(2,0,0)$

$x_1=1$

1.	$(0,-1,3)$
2.	$(0,\ 0,2)$
3.	$(0,\ 1,1)$
4.	$(0,\ 2,0)$
5.	$(1,-1,2)$
6.	$(1,\ 0,1)$
7.	$(1,\ 1,0)$
8.	$(2,-1,1)$
9.	$(2,\ 0,0)$
10.	$(3,-1,0)$

$\Rightarrow$

2.	$(0,0,2)$
3.	$(0,1,1)$
4.	$(0,2,0)$
6.	$(1,0,1)$
7.	$(1,1,0)$
9.	$(2,0,0)$

Note that in both cases the reduced truth table has entries with the same set of indexes. Therefore each subsequent party can apply a similar procedure to the table based on its own input, until the server obtains a table containing a single entry with the output of the function.

4.3 A Protocol for The Sum Function

We saw in Section 2.1 that the sum function $f(x_1, \ldots, x_n) = \sum_{j=1}^{n} x_j$ has a simple minimum-disclosure decomposition. As we will see now, it also has a very simple optimally-private protocol. Note that this protocol works even in the case that the domain of the inputs x_j is large (and thus the protocol for symmetric functions over $\mathbb{Z}_c$ cannot be used).

The semi-honest case. The protocol uses additively homomorphic encryption over the same group in which we compute the summation. The first party chooses $n-1$ random strings $r_{1,2}, \ldots, r_{1,n}$ and encrypts the value $x + \sum_{j=2}^{n} r_{1,j}$ under a public key pk_S that was chosen by the server. It also encrypts each $r_{1,j}$ under the public-key pk_j of the jth party P_j; denote $c_1^* = E_{pk_S}(x + \sum_{j=2}^{n} r_{1,j})$ and $c_{1,j} = E_{pk_j}(r_{1,j})$. Next, when party P_j contacts the server, it receives the ciphertexts c_{j-1}^* and $c_{i,j}$ for $i \leq j$, decrypts the ciphertexts $c_{i,j}$ in order to obtain the $r_{i,j}$'s. Party j chooses random $r_{j,k}$'s for $k > j$ and adds $x_j - \sum_{i<j} r_{i,j} + \sum_{k>j} r_{j,k}$ to c_{j-1}^* using the homomorphic operation. This updated ciphertext c_j^* is then sent back to the server, together with the $c_{j,k}$'s. It is clear that the only thing that can be learned by the server and a coalition of dishonest players is the partial sum of the honest parties up until the last honest party. Thus, this achieves the required level of security.

It is possible to implement this using Paillier's encryption [21]; this results in computing the sum modulo N. If it is necessary to compute the sum over the integers, then one simply needs to take N to be larger than $n \cdot L$ where (the absolute value of) each parties' input is an integer between 0 and L. Alternatively we can use LWE-based schemes that can be made additively homomorphic over any group $\mathbb{Z}_c$, e.g. [10]. (In this case we must take care to ensure that the distributions of evaluated ciphertexts do not leak information.)

PROTOCOL 4.6. (Semi-Honest Optimal Protocol for the Sum Function)

- **Inputs:** Each party P_i ($1 \leq i \leq n$) has a private input x_i, its own private key sk_i, and a vector of public keys $(pk_1, \ldots, pk_n, pk_S)$; the server P_{n+1} has (pk_S, sk_S) and $(pk_1, \ldots, pk_n)$. Let $\mathcal{D}$ be the domain of the homomorphic encryption scheme with public key pk_S.
- **The protocol:**
  1. *Initialization:* Let c_0^* be an arbitrary encryption of zero under the Server's public key.
  2. *Interaction of clients $P_1, \ldots, P_n$ with the server:* For every $j = 1, \ldots, n$:
     (a) The server P_{n+1} sends to P_j the ciphertexts c_{j-1}^* and all the $c_{i,j}$'s for $i < j$. (If $j = 1$ then P_1 can compute c_0^* on its own and the $c_{i,j}$'s are an empty list.)
     (b) For all $i < j$, Party P_j decrypts $c_{i,j}$ with its own secret key, $r_{i,j} \leftarrow \mathsf{Dec}_{sk_j}(c_{i,j})$. It also chooses a random $r_{j,k}$ for $k = j+1, \ldots, n+1$ and encrypts it under the key of Party P_k, $c_{j,k} \leftarrow \mathsf{Enc}_{pk_k}(r_{j,k})$.
     Let s_{j-1} be the value encrypted in c_{j-1}^*. Party j then uses the homomorphic properties with public key pk_S to compute c_j^* as an encryption of $s_j = s_{j-1} + x_j - \sum_{i<j} r_{i,j} + \sum_{k>j} r_{j,k}$.
     (c) Party P_j sends back to P_{n+1} the ciphertexts c_j^* and all the $c_{j,k}$'s for $k > j$.
  3. *Concluding the computation:* P_{n+1} decrypts c_{n+1}^* and outputs the value.

The malicious case. As in Protocol 4.2, we need to make sure that each party behaves correctly. First, we need to ensure that the server's public key was generated correctly (to ensure that the homomorphic operations truly hide each parties' individual input). This can be carried out efficiently and non-interactively using [3]. Next, we need to make sure that each party updates c_j^* correctly. This is a proof that two sums of ciphertexts (under different keys) are equal, which for Paillier's encryption could be rather expensive. Finally, we need to know the secret key sk_S of the server (in case it is corrupted) in order to learn the new encrypted value (and the difference) thereby defining each x_j. However, this is already obtained via the proof in [3]. As in Protocol 4.4, the parties all also sign on their messages and proofs, and all of these are verified.

4.4 The Selection Functions

In this section, we construct an optimally-private protocol for the selection function $f(j, x_2, \ldots, x_n) = x_j$; i.e., where the selector is first. As we have seen in Section 2.2, the disclosure in this case is the least. Specifically, if the last honest party is after the selected party, then the only thing learned by the server is the selected value and not even its position. Otherwise, the position is learned, but nothing else. (Note that hiding the position is really the only interesting issue in this function, since otherwise it can be trivially solved by having the selector first announce who is selected and next having the selected party send its value.)

The semi-honest case. Our protocol is similar to the following 1-out-of-N (semi-honest) oblivious transfer protocol, using additively homomorphic encryption: The receiver, who wants to get the j'th value, generates N ciphertexts, all encrypting 1 except the j'th that encrypts a 0. Using the additive-homomorphism, the sender multiplies the ciphertexts by random numbers (a different random number for each ciphertext) and then adds his value x_i to the i'th ciphertext. When the receiver decrypts, it gets the j'th value intact and all other values are random.

Our setting is a little more complicated than the OT setting, since (a) the inputs are split between parties $P_2, \ldots, P_n$ rather than all belonging to one sender, and (b) the receiver in our case is the server P_{n+1}, while the selection index j is known to the first party P_1. The latter concern is handled by choosing an encryption scheme with plaintext space much much larger than the domain of inputs to the parties. Now with high probability the j'th entry will be the only one in the domain of inputs, so the server can identify it.[5] To handle the first concern we will use a mix-net-like construction (using a layer-rerandomizable ecnryption), with each party shuffling the ciphertexts so that the following parties cannot tell which ciphertext came from what party. (Also, we use El Gamal which is multiplicative- rather than additive-homomorphic, so we modify the underlying OT protocol accordingly.)

In more detail, P_1 with selector input j prepares a vector of El Gamal ciphertexts, all encrypting the group generator G except the j'th that encrypts the group element 1. The i'th ciphertext in this vector is encrypted under the compound El Gamal public key $H_{i,n+1} = \prod_{t=i}^{n+1} h_t$. (When using a generic layer-rerandomizable encryption, the i'th ciphertext is encrypted onion-style under the public keys of parties i though $n + 1$.) We call this vector the "initial ciphretexts" and denote it by $\mathcal{I}$. During the protocol the initial ciphertexts will be passed unchanged, and the parties use them to process another vector of ciphertexts that contain the actual values. We call that other vector of ciphertexts the "work ciphertexts", and denote it by $\mathcal{W}$.

Each party P_i ($i \geq 2$) gets the initial ciphertexts $\mathcal{I}$ and a vector $\mathcal{W}_{i-1}$ of $i-2$ ciphertexts. The ciphretexts in $\mathcal{W}$ are all encrypted under $H_{i,n+1}$. P_i takes the i'th ciphertext from $\mathcal{I}$ (which is also encrypted under $H_{i,n+1}$), uses the multiplicative homomorphism of El Gamal to raise the plaintext inside it to a random power in $\mathbb{Z}_q$, then uses the homomorphism again to multiply the plaintext by its input x_i. It inserts the resulting ciphertext to $\mathcal{W}_{i-1}$, thus getting a vecotr of $i-1$ ciphertexts which we denote by $\mathcal{W}'_i$. P_i then peels off its layer of encryption (resulting in ciphertexts under $H_{i+1,n+1}$), randomly permutes the ciphertxts and re-randomizes them, thus obtaining a new vector of ciphertexts $\mathcal{W}_i$, which P_i sends back to the server.

After all the players participated, the server has a vector of "work ciphertexts" $\mathcal{W}_n$, encrypted under the public key of the server $H_{n+1} = h_{n+1}$. The server decrypts this vector, and if the corresponding plaintext vector has a single element from the input domain of the protocol then the server outputs that element. A pseudocode description of this protocol (described using a generic additively homomorphic encryption layer-rerandomizable) can be found in Protocol 4.7.

Using similar arguments as in the binary symmetric case, we have that Protocol 4.7 is optimally-private in the presence of augmented semi-honest adversaries, if the encryption schemes used is additively homomorphic and layer rerandomizable, and has plaintext space which is super-polynomially larger than the input space for the protocol.

An error-free variant. The small probability probability of error in the protocol above can be easily removed. If the input space for the protocol is some $\mathbb{Z}_c$, then we choose encryption scheme

[5] See remark at the end of this section for an error-free variant of the same idea.

PROTOCOL 4.7. (Semi-Honest Optimal Protocol for the Selection Function)

- **Inputs:** Party P_1 has an index j $(2 \le j \le n)$, and each party P_i $(2 \le i \le n)$ has a private input x_i, its own private key sk_i, and a vector of public keys $(pk_2, \ldots, pk_n, pk_{n+1})$.
- **The protocol:**
  1. *First party instructions:*
     (a) For every $i = 2, \ldots, n$, $i \ne j$, P_1 computes $c_i = \bar{E}_{pk_i,\ldots,pk_{n+1}}(1)$. For $i = j$, P_1 computes $c_j = \bar{E}_{pk_j,\ldots,pk_{n+1}}(0)$.
     (b) P_1 sends the vector of initial ciphretexts $\mathcal{I} = (c_2, \ldots, c_n)$ to the server P_{n+1}.
  2. *Interaction of clients $P_2, \ldots, P_n$ with server.* For $i = 2, \ldots, n$:
     (a) P_{n+1} sends P_i the initial ciphertexts $\mathcal{I}$, and a vector $\mathcal{W}_{i-1}$ of $i-2$ ciphertexts, encrypted under $\vec{\mathsf{pk}}_i = (pk_i, \ldots, pk_{n+1})$. (For $i = 2$, $\mathcal{W}_1$ is empty.)
     (b) P_i extracts the i'th ciphretext from $\mathcal{I}$, $c_i = \mathcal{I}[i]$ (encrypting a bit $b_i \in \{0,1\}$ under $\vec{\mathsf{pk}}_i$.) It chooses a random number r_i from the plaintext space and uses the encryption additive-homomorphism to compute a ciphertext $c'_i = r_i \boxtimes c_i \boxplus x_i$, encrypting the plaintext value $r_i \cdot b_i + x_i$.
     (c) P_i adds c'_i to the vectos $\mathcal{W}_{i-1}$ (thus receiving a vector of $i-1$ ciphertexts under $(pk_i, \ldots, pk_{n+1})$) and decrypts a layer of all of these ciphertexts using its secret key sk_i; denote the result by $\mathcal{W}'_i$.
     (d) P_i permutes the ciphertexts in $\mathcal{W}'_i$ and rerandomizes all of them using the public keys $pk_{i+1}, \ldots, pk_{n+1}$. Denoting the result by $\mathcal{W}_i$, P_i sends $\mathcal{W}_i$ back to the server.
  3. *Concluding the computation:* Upon receiving the encrypted vector $\mathcal{W}_n$ (of length $n-1$) from P_n, the server P_{n+1} decrypts all the ciphertext using its secret key sk_{n+1}. If the corresponding plaintext vector includes a single element from the input space then the server outputs that plaintext (else it outputs '?').

with plaintext space that includes $\mathbb{Z}_{c+1} = \{0, 1, \ldots, c\}$. Then party P_i $(i \ge 2)$ with input $x_i \in \mathbb{Z}_c$, instead of choosing r_i at random as above smiply sets $r_i = c - x_i$. This ensures that the value that the server recovers is either x_i (if P_i received an encryption of 0) or c (if P_i received an encryption of 1).

The malicious case. As above, in this case we need to have the parties prove that they behaved honestly. This can be achieved using similar techniques as those described above.

5 Securely Computing any Decomposition

We now turn to the task of securely computing an arbitrary given decomposition. For this we use *re-randomizable garbled circuits* that were introduced by Gentry et al. for the purpose of multi-Hop homomorphic encryption [11]. (Below we call this the GHV construction.) Very roughly, each party i receives from the server a garbled circuit encoding $\tilde{f}_{i-1}(x_1, \ldots, x_{i-1})$, adds its input to generate a garbled circuit for $\tilde{f}_i(x_1, \ldots, x_i)$, then re-randomizes this garbled circuit (so as to hide x_i from colluding dishonest parties $i-1$ and $i+1$) and sends the result back to the server.

The main problem that arises is that in our setting we do not want the server to be able to evaluate all the $\tilde{f}_i$'s. More specifically, if i is the index of the last honest party then we do not want

the adversary to be able to evaluate $\tilde{f}_j$ for any $j < i$. (In contrast, in the setting of multi-Hop homomorphic encryption if party $i+1$ is dishonest then the adversary can evaluate $\tilde{f}_i$.)

To solve this we again use layered re-randomizable encryption: instead of giving the players the input labels for the garbled circuit, we give them only the encryption of these input labels, encrypted under all the keys of of the parties that did not participate yet. Each party peels of its layer of encryption and re-randomizes the result, hence the server learns the input label only after all the (honest) parties decrypted their layers, and it cannot evaluate the circuit earlier.

We note that the layered re-randomizable encryption is intertwined with the garbled-circuit construction, since each party has to be able to transform the encryption of the inputs of one garbled circuit into "freshly random" encryption on the inputs to a re-randomized version of the garbled circuit. Recall that in the GHV construction the labels on the wires are balanced bit-strings (with half 0s and half 1s), and re-randomizing a circuit is done by bitwise permuting the labels. Hence we use bit-wise encryption (to handle the permutation) where ciphertexts can be re-randomized (to hide the correlation to the previous circuit).

The GHV construction is described in Section 5.1. We mention that the original construction from [11] is secure only in the semi-honest model. In particular a malicious party can choose "bad labels" to wires to foil re-randomization, by choosing the two labels on a wire with a very small (or very large) Hamming distance. We thus modify the construction slightly and require that the Hamming weight between the two labels be exactly half their length. This turns the GHV construction into one that works for any adversarial coins in the semi-honest model, so we can add (non-interactive) zero-knowledge proofs and get resilience against malicious adversary.

5.1 The GHV Construction

The GHV construction works over an algebraic group of prime order q where the decision Diffie-Hellman assumption is believed to hold. The labels on the wires of garbled circuits are of length $\ell = 3|q|$ (and we assume for convenience that ℓ is divisible by 4). A Boolean circuit is garbled by choosing two random ℓ-bit labels, each with Hamming weight exactly $\ell/2$, and in our variant also with Hamming distance exactly $\ell/2$ from each other. One way to choose such labels is to start with the labels $L0^* = 0^{\ell/2}1^{\ell/2}$ and $L1^* = 0^{\ell/4}1^{\ell/4}0^{\ell/4}1^{\ell/4}$, and then chose a random bit permutation π and apply it to both $L0^*, L1^*$, setting $L0 = \pi(L0^*)$ and $L1 = \pi(L1^*)$.

Given the labels on all the wires, a gate is represented by four pairs of ciphertexts under the (bit-by-bit version of the) BHHO cryptosystem [2], one pair for each line in the truth table of the gate. Let $R0, R1$ be the two labels on the first input wire, $S0, S1$ be the two labels on the second input wire, and $T0, T1$ be the two labels on the output wire of the gate. For a line $(a, b) \mapsto c$ in the truth table (with $a, b, c \in \{0, 1\}$), we choose at random $r, r' \in \{0, 1\}^\ell$ such that $r \oplus r' = Tc$, and encrypt r under Ra and r' under Sb, thus forming the pair $(\mathsf{Enc}_{Ra}(r), \mathsf{Enc}_{Sb}(r'))$. The gate is represented by the four pairs of ciphertexts in random order.

In addition, for the purpose of re-randomization we include with each ciphertext $\mathsf{Enc}_L(r)$ also the BHHO public key corresponding to the secret key L. The same public keys can be used also to identify the right ciphertext-pair to decrypt when evaluating the garbled circuit: If we know the two input labels Ra, Sb for some gate, then we decrypt the ciphertext-pair that includes the public keys corresponding to these two labels.

Finally, the mapping of the output labels of the circuit to 0-1 is done not by giving the labels themselves, but by giving the corresponding BHHO public keys. Namely, if the two labels on an output wire are $L0, L1$, corresponding to 0,1, respectively, then we include with the circuit two

BHHO public keys $\mathsf{pk}(L0)$ and $\mathsf{pk}(L1)$ in order. Upon evaluation, we output 0 if the output labels that we learn corresponds to the first public key and 1 if it corresponds to the second. (We note that these public keys allow one to extend the garbled circuit, since to generate $\mathsf{Enc}_L(r)$ it is sufficient to know $\mathsf{pk}(L)$ and we don't need to know L explicitly.)

Re-randomization of a circuit is done as follows:

1. Choose a random bit permutation π_w over $[\ell]$ for every wire w in the circuit. If the labels on that wire are $L0_w, L1_w$ then the new labels will be $\pi_w(L0_w), \pi_w(L1_w)$, respectively.
2. Use the BHHO key- and plaintext-homomorphism to translate each ciphertext $\mathsf{Enc}_L(r)$ into $\mathsf{Enc}_{\pi(L)}(\pi'(r))$ according to the permutations from Step 1. Also transform similarly all the attached public keys from $\mathsf{pk}(L)$ to $\mathsf{pk}(\pi(L))$. [6]
3. For each ciphertext-pair $(E_L(r), E_{L'}(r'))$, choose a random $\delta \in \{0,1\}^\ell$ and use the BHHO plaintext homomorphism to transform the pair into $(E_L(\delta \oplus r), E_{L'}(\delta \oplus r'))$.
4. Using the attached public keys, re-randomize all the BHHO ciphertexts and public keys, thus getting new (pseudo)random ciphertexts and public keys for the new labels.[7]

Let us denote by $\Lambda_f = \mathrm{GC}(f, r)$ the garbled circuit for circuit f that was generated using the randomness r, and by $\Lambda'_f = \mathrm{ReRand}(\Lambda_f, r^*)$ the re-randomized circuit that was generated from Λ_f using randomness r^*. The next lemma asserts that Λ'_f results is (pseudo)random, even conditioned on the randomness r.

Lemma 5.1. *If the decision Diffie-Hellman problem is hard in the group underlying the GHV construction, then for every fixed circuit f and randomness r, the following two distributions are computationally indistinguishable:*

$$\mathcal{D}_{\text{fresh}} \stackrel{\text{def}}{=} \left\{(f, r, \Lambda'_f) \;:\; r' \textit{ is fresh randomness}, \Lambda'_f = \mathrm{GC}(f, r')\right\}, \textit{ and}$$
$$\mathcal{D}_{\text{reRnd}} \stackrel{\text{def}}{=} \left\{(f, r, \Lambda'_f) \;:\; r^* \textit{ is fresh randomness}, \Lambda_f = \mathrm{GC}(f, r), \Lambda'_f = \mathrm{ReRand}(\Lambda_f, r^*)\right\}$$

Proof (sketch). This lemma was essentially proved in Theorem 7 of Gentry et al. [11] (Theorem 8 in the full version). The only difference is that it was only proved for randomly chosen r, not every r as stated above. The reason is that the GHV permutation lemma from [11] only applies when the two labels on a wire are chosen at random. We note however that the proof of that lemma goes through as long as the two labels on each wire have Hamming distance close to $\ell/2$ (and the only use of the randomly-chosen labels is to argue that their Hamming distance is indeed close to $\ell/2$). In our version of the construction the distance between the two labels is always exactly $\ell/2$, regardless of the randomness used. Hence we get the stronger statement of Lemma 5.1. □

5.2 Our Construction, Semi-Honest Model

As described above, we get security in our model by augmenting the GHV construction with encryption of the input labels. Differently from Gentry et al., we do not use oblivious transfer to

[6] BHHO public keys are just encryptions of 0, so every operation that can be applied to ciphertexts can also be applied to public keys.

[7] By pseudorandom we mean that they cannot be distinguished from fresh random ciphertexts (or public keys) even by someone who knows the corresponding secret keys.

encode the input of the first party but instead have that party encrypt the labels corresponding to its input bits with El Gamal. (We note that the same simplification could be used also in the contexts of multi-Hop homomorphic encryption.)

In more detail, our construction works in the PKI model, where each party i has a secret key sk_i and a corresponding public key $\mathsf{pk}_i = \mathsf{pk}(\mathsf{sk}_i)$, and every party knows the public keys of all other parties. In the description below we assume that these are all keys for El Gamal encryption, namely we have $\mathsf{sk}_i = \alpha_i \in \mathbb{Z}_q$ and $\mathsf{pk}_i = G^{\alpha_i}$ where G is a generator in an order-q group $\mathbb{G}$ in which DDH is hard.[8]

5.2.1 The Protocol

Let $\langle f_1, \ldots, f_n \rangle$ be a given decomposition that we want to implement. Namely, we want a protocol where the view of any set of cooperating semi-honest parties can be simulated knowing only the value $y_i = \tilde{f}_i(x_1, \ldots, x_i) = f_i(\ldots f_1(x_1), \ldots, x_i)$, where i is the index of the last honest party (i.e., the last party not in the set).

Throughout the computation, we maintain the invariant that before interacting with party i the server keeps a garbled circuit of the function $\tilde{f}_{i-1}(x_1, \ldots, x_{i-1})$ and an encryption of all the labels corresponding to the inputs bits in $x_1, \ldots, x_{i-1}$, where the encryption is with respect to the public keys of the remaining parties $\mathsf{pk}_i, \ldots, \mathsf{pk}_n, \mathsf{pk}_{n+1}$ (pk_{n+1} is the key of the server.)

In more detail, let Λ_{i-1} be a garbled circuit that the server keeps before talking to party i (where Λ_0 is the empty garbled circuit with no inputs). To slightly simplify notations we assume that all the inputs x_i are exactly t-bit long, and let x_{ij} denote the jth bit of x_i, i.e. $x_i = x_{i1}x_{i2} \ldots x_{it}$. Hence Λ_{i-1} has $(i-1)t$ input wires, and there are two ℓ-bit labels associated with each input wire. We denote the 0 and 1 labels associated with the jth input wire of the ith party by $L0_{ij}, L1_{ij}$, respectively.

Below we also denote by σ_{ij}^k the kth bit of the label corresponding to the input bit x_{ij}. That is, if $x_{ij} = 0$ then $(\sigma_{ij}^1 \ldots \sigma_{ij}^\ell) = L0_{ij}$ and if $x_{ij} = 1$ then $(\sigma_{ij}^1 \ldots \sigma_{ij}^\ell) = L1_{ij}$. Hence before talking to party P_i the server should have encryptions of all the bits $\sigma_{i'j}^k$ for $i' < i$, $j = 1, \ldots, t$ and $k = 1, \ldots, \ell$. Specifically, let H_i be the compounded public key of parties i through $n+1$, namely

$$H_i \stackrel{\text{def}}{=} \prod_{j=i}^{n+1} h_i.$$

Then for each bit $\sigma_{i'j}^k$ with $i' < i$, $j \leq t$ and $k \leq \ell$, the server keeps an El Gamal encryption of $\sigma_{i'j}^k$ relative to public key H_i, i.e., a pair of the form $(G^r, G^{\sigma_{ij}^k} \cdot {H_i}^r)$. (Of course, the exponents r in all these ciphertexts are chosen independently.)

The ith party. The ith party has its input $x_i = (x_{i1} \ldots x_{it})$, its secret key α_i and the public keys of the parties after it, $h_{i+1}, \ldots, h_n, h_{n+1}$. It receives from the server the garbled circuit Λ_{i-1} corresponding to $\tilde{f}_{i-1}$, and the encryption of all the bits σ_{ij}^k relative to the compounded public key H_i. Recall that $\tilde{f}_i$ is an extension of $\tilde{f}_{i-1}$ via $f_i(y_{i-1}, x_i)$, namely

$$\tilde{f}_i(x_1, \ldots, x_{i-1}, x_i) \;=\; f_i(\; \underbrace{\tilde{f}_{i-1}(x_1, \ldots, x_{i-1})}_{y_{i-1}} \;, x_i).$$

[8] In general one can use any bit-by-bit additively homomorphic encryption scheme instead of El Gamal, the details will be given in the full version.

Hence party P_i can extend the garbled circuit Λ_{i-1} corresponding to $\tilde{f}_{i-1}$ into a garbled circuit Λ_i corresponding to $\tilde{f}_i$, using the output labels of Λ_{i-1} as input labels for the wires of y_{i-1} and choosing new input labels for the wires of x_i. That is, party P_i builds the Yao circuit for $\tilde{f}_i$, choosing random labels for all wires except for the input wires corresponding to the output of $\tilde{f}_{i-1}$; the garbled labels on the input wires are taken as the output labels for the wires of the received circuit. Thus, the two circuits are composed into one.

Next, party P_i uses its secret key α_i to convert all the El Gamal ciphertexts relative to H_i into encryption of the same bits relative to H_{i+1}. Namely, given a ciphertext $(u = G^r, v = G^\sigma \cdot {H_i}^r)$, Party i computes $v' = v/u^{\alpha_i}$ and outputs the ciphertexts (u, v'). This is indeed an encryption of the bit σ with respect to H_{i+1}, since $H_{i+1} = H_i/h_i = H_i/G^{\alpha_i}$ and therefore

$$v' \;=\; \frac{v}{u^{\alpha_i}} \;=\; \frac{G^\sigma \cdot {H_i}^r}{G^{r\alpha_i}} \;=\; G^\sigma \cdot \left(\frac{H_i}{G^{\alpha_i}}\right)^r \;=\; G^\sigma \cdot {H_{i+1}}^r.$$

Party i also encrypts the bits σ_{ij}^k of the labels corresponding to all of its input bits x_{ij}, relative to the compounded public key H_{i+1}.

At this point Party i holds the complete state as needed for the next step of the computation, and it only remains to re-randomize this state (so as to hide x_i). To this end, Party i applies the re-randomization procedure from Section 5.1 to the garbled circuit Λ_i to get a new garbled circuit Λ_i'. This includes in particular choosing a random permutation π_{ij} for the wire of every input bit x_{ij}. Party P_i permutes accordingly the vector of El Gamal ciphertexts for the bits on that wire $(\sigma_{ij}^1 \dots \sigma_{ij}^\ell)$, thus obtaining an encryption of the new input label for this wire. (All these encryptions are relative to the compound public key H_{i+1}.) Finally it re-randomize these encryptions by choosing for each ciphertext a new exponent r' and replacing the pair $\langle u = G^r,\ v = G^\sigma \cdot H_{i+1}^r \rangle$ with $u' = u \cdot G^{r'} = G^{r+r'}$ and $v' = v \cdot H_{i+1}^{r'} = G^\sigma \cdot H_{i+1}^{r+r'}$. Party i sends Λ_i' and all the ciphertexts (in order) to the server, and the server is now ready for Party $i+1$.

The server. After the interaction with the last Party n, the server has a garbled circuit for the function $\tilde{f}_n = f$, and encryption of the input labels corresponding to all the input bits of all the parties, relative to the "compound" public key $H_{n+1} = h_{n+1}$. Since the server knows the secret key α_{n+1} corresponding to h_{n+1}, it can decrypt all these ciphertexts and recover the label on each input wire. The server then evaluates the garbled circuit and obtains the result $f(x_1, \dots, x_n)$, as needed.

5.3 Security in the Semi-Honest Model

Theorem 5.2. *For any decomposition $\bar{f} = \langle f_1, \dots, f_n \rangle$, the protocol from Section 5.2.1 is a server-based one-pass protocol that securely computes $\bar{f}$ in the semi-honest model, even if the dishonest players can chose arbitrary random coins for the protocol.*

Proof (sketch). Fix a given decomposition $\bar{f} = \langle f_1, \dots, f_n \rangle$ and a vector of inputs $\bar{x} = \langle x_1, \dots, x_n \rangle$. Also fix a subset of parties $S \subsetneq [n+1]$ and let i^* be the highest index not in S, $i^* = \max([n+1] \setminus S)$. We need to simulate the joint view of the parties in S (in the PKI model) given their own inputs $\langle x_i :\ i \in S \rangle$ and the value $y_{i^*} = \hat{f}_{i^*}(x_1, \dots, x_{i^*})$, and also the public keys of the honest parties $\langle h_i : i \notin S \rangle$.

The intuition behind the proof is that before the turn of party i^* the input labels of honest parties remain hidden since they are encrypted under a compound key that includes the public key

h_{i^*}. Hence essentially the only thing that the semi-honest set S sees is a set of garbled circuits for the functions $\tilde{f}_i$, $i = 1, \ldots, i^* - 1$ *without their input labels*, and these completely hide all the inputs of the honest parties. Moreover, re-randomizing the garbled circuit ensures that the parties in S cannot learn anything by correlating the garbled circuits for the different $\tilde{f}_i$'s. Only after party i^* uses its secret key to decrypt the input labels, does the set S learn a garbled circuit for $\tilde{f}_{i^*}$ together with one label on each input wire, and then the parties in S can compute y_{i^*}.

The simulator chooses at random the coins for the semi-honest parties $i \in S$ (including their secret and public keys), and gets the public keys of the honest parties, the inputs of the semi-honest parties, and if $i^* < n$ then also the value $y_{i^*} = \hat{f}_{i^*}(x_1, \ldots, x_{i^*})$.

The simulator works by generating for every honest party before i^* (i.e., $i \notin S, i < i^*$) a fake garbled circuit $\tilde{\Lambda}_i$ that has the same topology as $\tilde{f}_i$ but where all ciphertext-pairs in all the gates encrypt the all-zero label (instead of encrypting on of the labels on the output wire of the gate). The output mapping of these fake garbled circuits are chosen at random, and the simulator also produces fake encryptions of zero relative to the compounded key H_{i+1}, instead of the encryptions of labels on input wires.

If the server is semi-honest (i.e, $n + 1 \in S$ and therefore $i^* \leq n$), then the simulator also generates a fake garbled circuit $\tilde{\Lambda}_{i^*}$ that has the same topology as $\tilde{f}_{i^*}$ but where all the gates compute a constant function. (I.e., all input label combinations encrypt the same output label.) The output mapping of $\tilde{\Lambda}_{i^*}$ is set to be consistent with y_{i^*}, namely the label that is encrypted in gates leading to output wires is mapped to value of the corresponding bit of y_i. Also for this garbled circuit the simulator produces encryptions of the bits from one of the two labels on each input wire, relative to the compounded key H_{i^*+1}. (The label to encrypt is chosen at random from the two labels on that input wire.)

If the server is semi-honest ($n + 1 \in S$) then the view of the parties in the set S includes the fake garbled circuits and ciphertexts for all $i \notin S$. If the server is honest ($n + 1 \notin S$), the view of the parties in S includes only these fake garbled circuits and ciphertexts corresponding to parties i such that $i + 1 \in S$. The output of the simulator is what the semi-honest adversary $\mathcal{A}$ would have output on this view.

To prove that the simulation is indistinguishable from the real world, we go through several hybrids. Starting from the real-world, we first replace all the re-randomized garbled circuits computed by honest parties with fresh random garbled circuits for the same functions. By Lemma 5.1 these hybrids are indistinguishable from the real-world game. Then we replace the encryptions of the input wires of Λ_i ($i < i^*, i \notin S$) with encryptions of zeros, and use DDH to argue indistinguishability (since the parties in S do not know the secret key of party i^*). Next we setup hybrids similar to the proof of security for Yao circuits in [17], where we go over the gates from bottom to top and replace the encryptions of the "right labels" with encryptions of garbage. (For $i < i^*$ this garbage is the all-zero string, and for i^* the garbage is an encryption of one of the two labels on the output wire.) Here we argue indistinguishability by reduction to the semantic security of BHHO encryption in the gates. Finally we replace the output mapping for circuits $\tilde{\Lambda}_i$ $i < i^*$ with a random mapping, and here we argue that the view remains exactly the same (since only zeros are encrypted in any of the ciphertexts in these $\tilde{\Lambda}_i$'s). □

Remark. Note that when $i^* = n+1$ (i.e., the server is honest) then the view of S can be simulated knowing nothing but the inputs of the parties in S. This is because in out model only the server learns the output $f(x_1, \ldots, x_n)$. Similarly if $i^* = n$ (i.e., the server is semi-honest but the last party

is honest) then the view of S can be simulated knowing only the inputs of the parties in S and the output value $f(x_1, \ldots, x_n)$.

5.4 The Malicious Model

As we saw in Theorem 5.2, the security of the semi-honest protocol holds even if dishonest parties are allowed to choose their coins arbitrarily (rather than at random). Thus, to achieve security in the presence of malicious adversaries, it is sufficient that each party proves that it followed the instructions of the protocol relative to *some* input and set of random coins. This proof must be non-interactive and verified by all parties, and so a common reference string must be assumed. In order for us to extract the inputs used in the ideal-model simulation, the proof also has to be a proof of knowledge. One option for this is to use a universally composable non-interactive system of zero-knowledge proofs of knowledge, using enhanced trapdoor permutations [18].

In addition, we need to ensure that if the server is corrupted, then it does not modify any of the constructions carried out by the honest parties. This can be achieved using digital signatures (and having the signing key be part of the public-key infrastructure).

Theorem 5.3. *Assume the existence of enhanced trapdoor permutations and that DDH holds. Then, for any decomposition $\bar{f} = \langle f_1, \ldots, f_n \rangle$, there exists a server-based one-pass protocol that securely computes $\bar{f}$ in the semi-honest model, with a public-key infrastructure and in the common reference string model.*

6 Extensions and Open Problems

In this work we considered a very simple setting of a server and n clients that all know about each other (and in particular have each other's public keys), and where the order in which the clients connect to the server is pre-set. Our practical solutions for symmetric functions extend also to the "first come first serve" setting with no pre-set order, but for functions that are not symmetric even defining what security means may be nontrivial.

Another possible extension deals with functions that have natural "projection" on any subset of their variables. (For example, for the AVERAGE function, it is natural to talk about the average of any subset of the variables.) In this case, it may be desirable that the server be able to compute the function value as soon as at least t of the n clients connected to it.[9] Although it may be possible to replace the onion-like encryption in our protocols with encryption in a t-out-of-n manner, it seems nontrivial to do it in such a way that will still not allow a subset of t players to decrypt all the transcript of the protocol.

Another very interesting extension is when we have a large universe of clients that do not have each other's public keys, and we want to compute some function as soon as n of them connect to the server (e.g., polling). In this case it may be reasonable to assume that the clients all share some system parameters, and maybe even that each client has some secret key for the system, so perhaps tools from identity-based cryptography can be used here.

Finally, we point out that if we can get each of the players to connect twice to the server (rather than once), then our protocols can be used for achieving the standard notion of privacy

[9] In general, if we have a decomposition of f then we can think of $\tilde{f}_t(x_1, \ldots, x_t)$ as the projection of f on the first t variables. Computing $\tilde{f}_i$ may or may not be desirable, depending on the application.

for secure computation. Indeed, instead of computing the original n-input function $f(x_1, \ldots, x_n)$, we set up a protocol for computing the extended $2n$-input function that depends only the first n inputs $\hat{f}(x_1, \ldots, x_n, x_{n+1}, \ldots, x_{2n}) = f(x_1, \ldots, x_n)$. We consider a decomposition of $\hat{f}$ where the intermediate value after the n'th input is $f(x_1, \ldots, x_n)$, design a protocol to realize it, and let party i play the role of both parties i and $i+n$ in this protocol. With this setup, if even one of the players is honest then the intermediate result after "the last honest player" in the protocol is $f(x_1, \ldots, x_n)$, hence the view of the corrupted parties can be simulated knowing only this value.

References

[1] The Facebook Privacy Scandal. ABC News, October 19, 2010. `http://abcnews.go.com/GMA/Consumer/facebook-privacy-scandal-facebooks-watergate/story?id=11912201`.

[2] D. Boneh, S. Halevi, M. Hamburg, and R. Ostrovsky. Circular-Secure Encryption from Decision Diffie-Hellman. In *Advances in Cryptology - CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2008.

[3] J. Camenisch and M. Michels. Proving in Zero-Knowledge that a Number Is the Product of Two Safe Primes. In *Advances in Cryptology - EUROCRYPT 1999*, volume 1592 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 1999.

[4] S. G. Choi, A. Elbaz, T. Malkin, and M. Yung. Secure Multi-party Computation Minimizing Online Rounds. In M. Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 268–286. Springer, 2009.

[5] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols. In *Advances in Cryptology - CRYPTO 1994*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 1994.

[6] I. Damgård. On Σ Protocols. `http://www.daimi.au.dk/~ivan/Sigma.pdf`.

[7] A. Fiat and A. Shamir. How to Prove Yourself. Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology - CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–189. Springer-Verlag, 1986.

[8] Google Admits Weekend Privacy Breach. Fox News, March 10, 2009. `http://www.foxnews.com/story/0,2933,508324,00.html`.

[9] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing – STOC 2009*, pages 169–178. ACM, 2009.

[10] C. Gentry, S. Halevi, and V. Vaikuntanathan. A Simple BGN-type Cryptosystem from LWE. In *Advances in Cryptology - EUROCRYPT'10*, volume 6110 of *Lecture Notes in Computer Science*, pages 506–522. Springer, 2010. Full version available on-line from `http://eprint.iacr.org/2010/145`.

[11] C. Gentry, S. Halevi, and V. Vaikuntanathan. i-Hop Homomorphic Encryption and Rerandomizable Yao Circuits. In *Advances in Cryptology - CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2010. Full version available on-line from `http://eprint.iacr.org/2010/145`.

[12] O. Goldreich. *Foundations of Cryptography, Basic Applications*, volume 2. Cambridge University Press, 2004.

[13] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):210–217, 1986.

[14] D. Harnik, Y. Ishai, and E. Kushilevitz. How Many Oblivious Transfers Are Needed for Secure Multiparty Computation? In *Advances in Cryptology - CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 284–302. Springer, 2007.

[15] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby. A Pseudorandom Generator from any One-way Function. *SIAM J. Comput.*, 28(4):1364–1396, 1999.

[16] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer, November 2010.

[17] Y. Lindell and B. Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptology*, 22(2), 2009.

[18] A. De Santis, G. Di Crescenzo, R. Ostrovsky, G. Persiano, and A. Sahai. Robust Non-interactive Zero Knowledge. In J. Kilian, editor, *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 566–598, 2001.

[19] M. Naor and O. Reingold. Number-Theoretic Constructions of Efficient Pseudo-random Functions. *J. ACM*, 51(2):231–262, 2004.

[20] A Face Is Exposed for AOL Searcher No. 4417749. The New York Times, August 9, 2006. `http://www.nytimes.com/2006/08/09/technology/09aol.html`.

[21] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology - EUROCRYPT 1999*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[22] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–177. Academic Press, 1978.