Secure Computation on the Web: Computing without Simultaneous Interaction

Shai Halevi[1], Yehuda Lindell[2,⋆], and Benny Pinkas[2,⋆,⋆⋆]

[1] IBM T.J. Watson Research Center
shaih@alum.mit.edu
[2] Bar-Ilan University
lindell@biu.ac.il, benny@pinkas.net

Abstract. Secure computation enables mutually suspicious parties to compute a joint function of their private inputs while providing strong security guarantees. However, 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.

We initiate a 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.

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 newspaper web sites, bidders engaging in on-line auctions, voters using web-based election systems, and so on. In many cases, direct interaction between users is impossible simply because users are off line most of the 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

⋆ Supported by the European Research Council as part of the ERC project LAST.
⋆⋆ Supported in part by the Israel Science Foundation (grant No. 860/06).

P. Rogaway (Ed.): CRYPTO 2011, LNCS 6841, pp. 132–150, 2011.

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 example see [1].)

A natural approach toward rectifying this problem is to use cryptographic techniques 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., [2, 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 such mass deployment has not happened. Some of the reasons are related to practical issues with browser technology (e.g., clients cannot verify that they run the right program), but here we focus on a more cryptographic reason; specifically, the fact that our current multi-party protocols seem incompatible with the communication patterns of 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 strong communication primitives like broadcast. In contrast, clients on the web connect in an ad hoc manner via a server at different times, and typically do not communicate with each other. We thus ask whether one 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.

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 [3,4] does not solve the problem of secure computation in our setting. This is due to the fact that 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

We initiate a study of secure computation with loosely connected parties. 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 achieved 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 [5].

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.

We stress that although our basic model assumes a pre-set order, many of the protocols that we describe allow the parties to interact with the server in an arbitrary order, which need not be set up in advance. However, all our protocols assume that the clients connect to the server sequentially, removing this requirement is an interesting open problem.

Consider first the case of semi-honest parties. It is easy to see that even in this model protocols *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-pass decomposition* of a function: A decomposition of an n-input function $f(x_1, x_2, \ldots, x_n)$ is a vector of functions $\{f_i(y_{i-1}, x_i) : i = 1, \ldots, n\}$, such that for all inputs $x_1, \ldots, x_n$ it holds that $f(x_1, x_2, \ldots, x_n) = f_n(\cdots f_2(f_1(x_1), x_2) \ldots, x_n)$. Here y_i represents the *intermediate result* based on the inputs of parties P_1 through P_i and y_0 is defined as the empty string. 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.

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 (in the case of a corrupted server and corrupted party P_n the adversary learns all the inputs when the identity function is used, in contrast to a partial sum only).

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. For example, it is easy to see that for the sum function, having the f_i's be the partial sums is indeed a minimum-disclosure decomposition, because given $x_{i+1}, \ldots, x_n$ and the output y_n it is possible to compute the partial sum y_i. In Section 2 we define this notion of minimum-disclosure decompositions and describe many functions that have efficient minimum-disclosure decompositions. Then 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). Thus, for example, we construct a practical protocol for computing a referendum, as privately as is possible in our model.

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 P_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. (We remark that a weaker definition where bad parties can learn all the y_i's for which party P_{i+1} is dishonest, is essentially equivalent to the notion of i-Hop homomorphic encryption from [6].)

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. [6] we show that under the DDH assumption any efficient decomposition of f can be securely computed in our model (if a PKI is available). Our treatment simplifies the techniques from [6], in that we use re-randomizable garbled circuits only in conjunction with re-randomizable encryption (whereas [6] needed also re-randomizable OT). We also strengthen the construction from [6] 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 are similar to those used in the work of Harnik et al. [7]. In that work they considered a multi-party computation settings where the inputs of parties are incorporated one at a time, with the goal of minimizing the number of OTs that are needed every time a new input is received. In particular our protocols for symmetric functions are reminiscent of their tables method.

Another related work is that of Choi et al. [8]. 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 be 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.

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

A randomized one-pass decomposition of f is a sequence of n randomized functions with the same domains and ranges as above, such that the equality above 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,

$$\tilde{f}_i(x_1, x_2, \ldots, x_i) \stackrel{\text{def}}{=} f_i(\cdots f_2(f_1(x_1), x_2) \cdots, x_i). \tag{1}$$

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), while others do not. Fix an n-input function f and n particular inputs

$x_1, \ldots, x_n$. 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

$$g_i^{\vec{x}}(z_{i+1}, \ldots, z_n) \stackrel{\text{def}}{=} f(x_1, \ldots, x_i, z_{i+1}, \ldots, z_n). \tag{2}$$

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

We stress that not all functions have efficient minimum-disclosure decompositions,[2] as is stated in the following theorem.

Theorem 1. *If one-way functions exist, then there are functions that do not have efficient minimum-disclosure decompositions.*

The theorem is proved in the full version of the paper [9]. Roughly, a decomposition is minimum-disclosure only when the residual functions g_i are efficiently learnable. Hence, a pseudorandom function $f : \mathsf{Seeds} \times \mathsf{Inputs} \to \mathsf{Outputs}$ (when viewed as a two-input function $f(s, x)$) does not have an efficient minimum-disclosure decomposition. In the full version we also include a discussion about functions with incomparable decompositions.

2.2 Some 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 the decomposition into partial sums $f_i(y_{i-1}, x_i) = y_{i-1} + x_i$ is clearly 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: $g_i^{\vec{x}}(0, \ldots, 0) = f(x_1, \ldots, x_i, 0, \ldots, 0) = \sum_{j=1}^{i} x_i$.

Selection functions. Other illustrating examples of functions with minimum-disclosure decompositions are 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 $i = n$

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

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

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. 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 simulator can output x_j (which is the output of $g_i^{\vec{x}}$ on every input).

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 j. 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 imply 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.

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 $g_{i+1}^{\vec{x}}$ (see the full version 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 $g_i^{\vec{x}}$ 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. Namely, where 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 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. 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 $\mathrm{REAL}_{\bar{\pi},\mathcal{A}(z),I}(\vec{x}, \vec{kp}, 1^m)$.

Securely computing a decomposition. We define security via the ideal/real paradigm in the stand-alone setting with static corruptions. 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 of all clients and the identities of corrupted parties, and sends to the server the function output as well as any information that is inherently learned in our model (based on who is corrupted). Note 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. 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 $\mathrm{IDEAL}^{\mathrm{SH}}_{\bar{f},\mathcal{S}(z),I}(\vec{x}, z, 1^m)$.

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 $\mathrm{IDEAL}^{\mathrm{MAL}}_{\bar{f},\mathcal{S}(z),I}(\vec{x}, z, 1^m)$.

Definition 3 (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 $z \in \{0,1\}^*$

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

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. When the function has a minimum-disclosure decomposition and a protocol that realizes that decomposition, then that protocol is called *optimally-private*.

Definition 4 (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.*

4 Practical Optimal Protocols

In Section 5 we show that *any* decomposition can be securely computed given a public-key infrastructure, under the DDH assumption. As a corollary we obtain that 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.) In the full version we describe protocols for the other functions from Section 2.2.

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.

The Semi-Honest Case. Recall that symmetric functions have a concise truth table of size $n+1$, 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 carried out 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 a layered, or 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, or the last row of the truth table if its input is 1. After the last party, the table contains just one row which is encrypted under the server's key.

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. In addition, it can see all the ciphertexts received 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 5. *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* $\vec{r} \in (\{0,1\}^*)^n$,

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

where $\vec{pk} = (pk_1,\ldots,pk_n)$ *is such that all the* pk_i*'s are in the range of* G*, and* $\vec{s} \in_R (\{0,1\}^*)^n$ *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. Layer rerandomizability can be obtained from any additively homomorphic encryption scheme. Namely, define an initial layered encryption of x by

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

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

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

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.

[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.

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 transforms a $(j-1)$th level layered encryption to a jth level layered encryption.

A more 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 $H_{i,n} = \prod_{j=i}^{n} h_j = G^{\sum_{j=i}^{n} \alpha_j}$. 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 $u' = u$ and $v' = v \cdot u^{-\alpha_i}$. This works because taking $u = G^r$ and $v = x \cdot (H_{i,n})^r$ we have that

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

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 $v'' = v' \cdot (H_{i+1,n})^s$.

Returning to symmetric functions, in our protocol we will now use layer rerandomizable encryption to encrypt the lines of the truth table, and each party in turn will decrypt its own layer, remove either the first or last row from the table, rerandomize and then send back to the server.

Theorem 2. *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 the protocol above 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). □

In the protocol above, using the El-Gamal-based rerandomizable encryption, each party computes less than $3n$ exponentiations, so the total number of exponentiations if at most $3n^2$. Hence this protocol could be practical for a large (but not huge) number of parties, perhaps even for n in the thousands. We remark however that the parties must work sequentially, and this may be a limitation if n is too large. Also, in this concrete instantiation the parties can connect and interact with the server in any order, which is an important property for practical implementation and deployment.

The Malicious Case. Since the semi-honest protocol is secure for any random coins used by the dishonest parties, it is enough to add signatures so that corrupt parties and/or server cannot modify the messages sent by previous parties, and (non-interactive) zero-knowledge proofs of good behavior to obtain security in the malicious model. We describe the resulting protocol in the full version. For our concrete El Gamal implementation, all these proofs can be made efficient since they can all be reduced to compound statements about equality of discrete logarithms, and these can be made non-interactive using the Fiat-Shamir transformation in the random-oracle model. In particular, each party needs to compute $O(n^2)$ exponentiations, we estimate that running the protocol with $n = 100$ parties will take just a few minutes per party.

Symmetric Functions Over Larger Domains. In the full version we also show that the protocols for binary symmetric functions extend to symmetric functions over any domain $\mathbb{Z}_c$, where the complexity grows as $n^{O(c)}$. Hence we get efficient protocols for any constant c.

4.2 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 jth value, generates N ciphertexts, all encrypting 1 except the jth that encrypts a 0. Using the additive-homomorphism, the sender multiplies the ciphertexts by random scalars (a different random number for each ciphertext) and then adds its value x_i to the ith ciphertext. When the receiver decrypts, it gets the jth 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 larger than the domain of inputs to the parties. Now with high probability the jth entry will be the only one in the domain of inputs, so the server can identify it. To handle the first concern we will use a mix-net-like construction (using a layer-rerandomizable encryption), 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, and 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 for the jth entry that encrypts the group element 1. The ith 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 ith ciphertext is encrypted onion-style under the public keys of parties i though $n+1$.) We call this vector the "initial ciphertexts" 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 ($i \geq 2$) receives the initial ciphertexts $\mathcal{I}$ and a vector $\mathcal{W}_{i-1}$ of $i-2$ ciphertexts. The ciphertexts in $\mathcal{W}_{i-1}$ are all encrypted under $H_{i,n+1}$. P_i takes the ith 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 vector 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 ciphertexts 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 parties have 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 3.

PROTOCOL 3 (Semi-Honest Optimal Protocol for the Selection Function)

- **Inputs:** Party P_1 has an index j ($2 \leq j \leq n$), and each party P_i ($2 \leq i \leq n$) 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 $i = 2, \ldots, n$, $i \neq j$, 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 ciphertexts $\mathcal{I} = (c_2, \ldots, c_n)$ 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 ith ciphertext 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 additive-homomorphic property of the encryption to compute an encryption of $r_i \cdot b_i + x_i$, using $c_i = E_{\vec{\mathsf{pk}}_i}(b_i)$, r_i and x_i.
     (c) P_i adds c'_i to the vectors $\mathcal{W}_{i-1}$ (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 (otherwise it outputs '?').

Using similar arguments as in the binary symmetric case, we have that Protocol 3 is optimally-private in the presence of 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.

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

5 Securely Computing any Decomposition

In this section, we present a basic feasibility result regarding the possibility of securely computing an arbitrary given decomposition in our model. For this we use *re-randomizable garbled circuits* that were introduced by Gentry et al. for the purpose of multi-Hop homomorphic encryption [6]. (Below we call this the GHV construction.) Roughly, each party P_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 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 P_{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 parties the input labels for the garbled circuit, we give them only the encryption of these input labels, encrypted under all the keys 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 of 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).

We mention that the original construction from [6] 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 distance 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 adversaries.

5.1 Our Construction, Semi-Honest Model

As described above, we obtain 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 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 also be used in the context of multi-Hop homomorphic encryption.)

In more detail, our construction works in the PKI model, where each party P_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.

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 of corrupted parties).

Throughout the computation, we maintain the invariant that before interacting with party P_i the server has 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 has before talking to party P_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 $i' < i$, $j \le t$ and $k \le \ell$, the server has 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

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

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 P_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 P_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 P_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 P_i applies the re-randomization procedure 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 $(\sigma_{ij}^1 \ldots \sigma_{ij}^\ell)$, 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-randomizes 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 P_i sends Λ_i' and all the ciphertexts (in order) to the server, and the server is now ready for party P_{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 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 $f(x_1, \ldots, x_n)$, as needed. The proof of the following theorem can be found in the full version [9].

Theorem 3. *For any decomposition $\bar{f} = \langle f_1, \ldots, f_n \rangle$, the protocol from Section 5.1 is a server-based one-pass protocol that securely computes $\bar{f}$ in the semi-honest model, even if the dishonest parties can choose arbitrary random coins for the protocol.*

5.2 The Malicious Model

As we saw in Theorem 3, 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, we have each party prove 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 subsequent parties. This requires a common reference string (or perhaps re-use of the available PKI). 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 [10].

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 4. *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 malicious 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 still require all parties to know about each other. In addition, all our solutions are sequential, they all rely heavily on the fact that client i completes the interaction with the server before client $i+1$ begins. Allowing concurrency between clients is a very interesting open problem and may be crucial for a large number of clients.

Another possible extension deals with functions that have natural "projections" 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.[4] 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 parties to decrypt the entire 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.

[4] 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.

Finally, we point out that if we can have each of the parties connect twice to the server (rather than once), then our protocols can be used for achieving the standard notion of privacy 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 nth input is $f(x_1, \ldots, x_n)$, design a protocol to realize it, and let party P_i play the role of both parties i and $i + n$ in this protocol. With this protocol, as soon as one of the parties is honest we have that the intermediate result after "the last honest party" in the protocol is $f(x_1, \ldots, x_n)$. Hence the view of the corrupted parties can be simulated knowing only this value.

Acknowledgments. We thank the CRYPTO 2011 reviewers for their many helpful comments.

References

1. 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`
2. Goldreich, O.: Foundations of Cryptography, Basic Applications, vol. 2. Cambridge University Press, Cambridge (2004)
3. Rivest, R., Adleman, L., Dertouzos, M.: On data banks and privacy homomorphisms. In: Foundations of Secure Computation, pp. 169–177. Academic Press, London (1978)
4. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: 41st ACM Symposium on Theory of Computing – STOC 2009, pp. 169–178. ACM, New York (2009)
5. Fiat, A., Shamir, A.: How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (1987)
6. Gentry, C., Halevi, S., Vaikuntanathan, V.: i-Hop Homomorphic Encryption and Rerandomizable Yao Circuits. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 155–172. Springer, Heidelberg (2010), Full version available online at `http://eprint.iacr.org/2010/145`
7. Harnik, D., Ishai, Y., Kushilevitz, E.: How Many Oblivious Transfers Are Needed for Secure Multiparty Computation? In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 284–302. Springer, Heidelberg (2007)
8. Choi, S.G., Elbaz, A., Malkin, T., Yung, M.: Secure Multi-party Computation Minimizing Online Rounds. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 268–286. Springer, Heidelberg (2009)
9. Halevi, S., Lindell, Y., Pinkas, B.: Secure Computation on the Web: Computing without Simultaneous Interaction, `http://eprint.iacr.org/2011/157`
10. De Santis, A., Di Crescenzo, G., Ostrovsky, R., Persiano, G., Sahai, A.: Robust Non-interactive Zero Knowledge. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 566–598. Springer, Heidelberg (2001)