# Multi-Client Non-Interactive Verifiable Computation

Seung Geol Choi[1⋆], Jonathan Katz[2],
Ranjit Kumaresan[3⋆], and Carlos Cid[4]

[1] Department of Computer Science, Columbia University
sgchoi@cs.columbia.edu
[2] Department of Computer Science, University of Maryland
jkatz@cs.umd.edu
[3] Department of Computer Science, Technion
ranjit@cs.technion.ac.il
[4] Royal Holloway, University of London
carlos.cid@rhul.ac.uk

**Abstract.** Gennaro et al. (Crypto 2010) introduced the notion of *non-interactive verifiable computation*, which allows a computationally weak client to outsource the computation of a function $f$ on a series of inputs $x^{(1)}, \ldots$ to a more powerful but untrusted server. Following a pre-processing phase (that is carried out only once), the client sends some representation of its current input $x^{(i)}$ to the server; the server returns an answer that allows the client to recover the correct result $f(x^{(i)})$, accompanied by a proof of correctness that ensures the client does not accept an incorrect result. The crucial property is that the work done by the client in preparing its input and verifying the server's proof is less than the time required for the client to compute $f$ on its own.

We extend this notion to the *multi-client* setting, where $n$ computationally weak clients wish to outsource to an untrusted server the computation of a function $f$ over a series of *joint* inputs $(x_1^{(1)}, \ldots, x_n^{(1)}), \ldots$ without interacting with each other. We present a construction for this setting by combining the scheme of Gennaro et al. with a primitive called proxy oblivious transfer.

## 1 Introduction

There are many instances in which it is desirable to outsource computation from a relatively weak computational device (a *client*) to a more powerful entity or collection of entities (*servers*). Notable examples include:

– Distributed-computing projects (e.g., SETI@Home or distributed.net), in which idle processing time on thousands of computers is harnessed to solve a computational problem.

⋆ Portions of this work were done while at the University of Maryland.

– Cloud computing, where individuals or businesses can purchase computing power as a service when it is needed to solve some difficult task.
– Outsourcing computationally intensive operations from weak mobile devices (e.g., smartphones, sensors) to a back-end server or some other third party.

In each of the above scenarios, there may be an incentive for the server to try to cheat and return an incorrect result to the client. This may be related to the nature of the computation being performed — e.g., if the server wants to convince the client of a particular result because it will have beneficial consequences for the server — or may simply be due to the server's desire to minimize the use of its own computational resources. Errors can also occur due to faulty algorithm implementation or system failures. In all these cases, the client needs some guarantee that the answer returned from the server is correct.

This problem of *verifiable (outsourced) computation* has attracted many researchers, and various protocols have been proposed (see Section 1.3). Recently, Gennaro et al. [13] formalized the problem of *non-interactive* verifiable computation in which there is only one round of interaction between the client and the server each time a computation is performed. Specifically, fix some function $f$ that the client wants to compute. Following a pre-processing phase (that is carried out only once), the client can then repeatedly request the server to compute $f$ on inputs $x^{(1)}, \ldots$ of its choice via the following steps:

**Input preparation:** In time period $i$, the client processes its current input $x^{(i)}$ to obtain some representation of this input, which it sends to the server.
**Output computation:** The server computes a response that encodes the correct answer $f(x^{(i)})$ along with a proof that it was computed correctly.
**Output verification:** The client recovers $f(x^{(i)})$ from the response provided by the server, and verifies the proof that this result is correct.

The above is only interesting if the input-preparation and output-verification stages require less time (in total) than the time required for the client to compute $f$ by itself. (The time required for pre-processing is ignored, as it is assumed to be amortized over several evaluations of $f$.) Less crucial, but still important, is that the time required for the output-computation phase should not be much larger than the time required to compute $f$ (otherwise the cost to the server may be too burdensome). Gennaro et al. construct a non-interactive verifiable-computation scheme based on Yao's garbled-circuit protocol [25] and any fully homomorphic encryption scheme.

### 1.1 Our Results

The scheme presented in [13] is inherently *single-client*. There are, however, scenarios in which it would be desirable to extend this functionality to the *multi-client* setting, e.g., networks made up of several resource-constrained nodes (sensors) that collectively gather data to be used jointly as input to some computation. In this work we initiate consideration of this setting. We assume $n$

(semi-honest) clients wish to outsource the computation of some function $f$ over a series of *joint* inputs $(x_1^{(1)}, \ldots, x_n^{(1)}), \ldots$ to an untrusted server.

A trivial solution to the problem would be for the last $n-1$ clients to send their inputs to the first client, who can then run a single-client verifiable-computation scheme and forward the result (assuming verification succeeded) to the other clients. This suffers from several drawbacks:

- This solution requires the clients to communicate *with each other*. There may be scenarios (e.g., sensors spread across a large geographical region) where clients can all communicate with a central server but are unable to communicate directly with each other.[1]
- This solution achieves no *privacy* since the first client sees the inputs of all the other clients.

Addressing the first drawback, we consider only *non-interactive* protocols in which each client communicates only with the server. A definition of soundness in the non-interactive setting is subtle, since without some additional assumptions (1) there is no way for one client to distinguish another legitimate client from a cheating server who tries to provide its own input $x_i$, and (2) there is nothing that "binds" the input of one client at one time period to the input of another client at that same time period (and thus the server could potentially "mix-and-match" the first-period input of the first client with the second-period input of the second client). We address these issues by assuming that (1) there is a public-key infrastructure (PKI), such that all clients have public keys known to each other, and (2) all clients maintain a counter indicating how many times they have interacted with the server (or, equivalently, there is some global notion of time). These assumptions are reasonable and (essentially) necessary to prevent the difficulties mentioned above.

Addressing the second drawback, we also define a notion of privacy of the clients' input from each other that we require any solution to satisfy. This is in addition to privacy of the clients' inputs from the server, as in [13].

In addition to defining the model, we also show a construction of a protocol for non-interactive, multi-client verifiable computation. We give an overview of our construction in the following section.

### 1.2 Overview of Our Scheme

Our construction is a generalization of the single-client solution by Gennaro et al., so we begin with a brief description of their scheme.

**Single-client verifiable computation.** We first describe the basic idea. Let Alice be a client who wishes to outsource computation of a function $f$ to a server. In the pre-processing phase, Alice creates a garbled circuit that corresponds to $f$,

[1] Note that having the clients communicate with each other by routing all their messages via the server (using end-to-end authenticated encryption, say) would require additional rounds of interaction.

and sends it to the server. Later, in the online phase, Alice computes input-wire keys for the garbled circuit that correspond to her actual input $x$, and sends these keys to the server. The server evaluates the garbled circuit using the input-wire keys provided by Alice to obtain the output keys of the garbled circuit; if the server behaves honestly, these output keys correspond to the correct output $f(x)$. The server returns these output keys to Alice, who then checks if the key received from the server on each output wire is a legitimate output key (i.e., one of the two possibilities) for that wire. If so, then Alice determines the actual output based on the keys received from the server. Loosely speaking, verifiability of this scheme follows from the fact that evaluation of a garbled circuit on input-wire keys corresponding to an input $x$ does not reveal information about any output-wire keys other than those that correspond to $f(x)$. (See also [3].)

The scheme described above works only for a single evaluation of $f$. To accommodate multiple evaluations of $f$, Gennaro et al. propose the use of fully homomorphic encryption (FHE) in the following way. The pre-processing step is the same as before. However, in the online phase, Alice generates a fresh public/private key pair for an FHE scheme each time she wants the server to evaluate $f$. She then encrypts the input-wire keys that correspond to her input using this public key, and sends these encryptions (along with the public key) to the server. Using the homomorphic properties of the encryption scheme, the server now runs the previous scheme to obtain *encryptions* of the output-wire keys corresponding to the correct output. The server returns the resulting ciphertexts to Alice, who decrypts them and then verifies the result as before. Security of this scheme follows from the soundness of the one-time scheme described earlier and semantic security of the FHE scheme.

**Multi-client verifiable computation.** In the rest of the overview, we discuss how to adapt the solution of Gennaro et al. to the multi-client setting. In our discussion, we consider the case where only the first client gets output. (The more general case is handled by simply having the clients run several executions of the scheme in parallel, with each client playing the role of the first in one execution.) We discuss the case of two clients here for simplicity, but our solution extends naturally to the general case.

Suppose two clients Alice and Bob want to outsource a computation to a server. Applying a similar idea as before, say Alice creates a garbled circuit which it sends to the server in the pre-processing phase. During the online phase, Alice will be able to compute and send to the server input-wire keys that correspond to her input. However, it is unclear how the server can obtain the input-wire keys corresponding to Bob's input. Recall that we are interested in a *non-interactive* solution and Alice does not know the input of Bob; moreover, Alice cannot send two input-wire keys for any wire to the server or else soundness is violated.

We overcome this difficulty using a gadget called *proxy oblivious transfer* (proxy OT) [22]. In proxy OT, there is a *sender* that holds inputs $(a_0, a_1)$, a *chooser* that holds input bit $b$, and a *proxy* that, at the end of the protocol, learns $a_b$ and nothing else. Since we are ultimately interested in a non-interactive solution for multi-client verifiable computation, we will be interested only in non-

interactive proxy-OT schemes. We show a construction of a non-interactive proxy OT from any non-interactive key-exchange protocol.

Coming back to the multi-client verifiable-computation scheme described earlier, we can use proxy OT to enable the server to learn the appropriate input-wire label for each wire that corresponds to Bob's input. In more detail, let $(\widetilde{w}_0, \widetilde{w}_1)$ denote the keys for input-wire $w$ in the garbled circuit that was created by Alice in the pre-processing phase. Alice acts as the sender with inputs $(\widetilde{w}_0, \widetilde{w}_1)$ in a proxy OT protocol, and Bob acts as the chooser with his actual input bit $b$ for that wire. The server is the proxy, and obtains output $\widetilde{w}_b$. The server learns nothing about $\widetilde{w}_{1-b}$ and so, informally, soundness is preserved. The rest of the protocol proceeds in the same way as the single-client protocol. The extension to accommodate multiple evaluations of $f$ is done using fully homomorphic encryption as described earlier.

**A generic approach to multi-client outsourcing.** It is not hard to see that our techniques can be applied to any single-client, non-interactive, verifiable-computation scheme that is *projective* in the following (informal) sense: the input-preparation stage generates a vector of pairs $(w_{1,0}, w_{1,1}), \ldots, (w_{\ell,0}, w_{\ell,1})$, and the client sends $w_{1,x_1}, \ldots, w_{\ell,x_\ell}$ to the server.

### 1.3 Related Work

The problems of outsourcing and verifiable computation have been extensively studied. Works such as [9, 16, 17] have focused on outsourcing expensive cryptographic operations (e.g., modular exponentiations, one-way function inversion) to semi-trusted devices. Verifiable computation has been the focus of a long line of research starting from works on interactive proofs [2, 15], and efficient argument systems [19, 21, 20]. In particular, Micali's work [21] gives a solution for non-interactive verifiable computation in the random oracle model. Goldwasser, Kalai, and Rothblum [14] give an interactive protocol to verify certain computations efficiently; their solution can be made non-interactive for a restricted class of functions.

Gennaro et al. [13] formally defined the notion of non-interactive verifiable computation for general functions and gave a construction achieving this notion. Subsequent schemes for non-interactive verifiable computation of general functions include [10, 1]. Other works have focused on improving the efficiency of schemes for verifiable computation of specific functions [5, 24, 12, 23], or in slightly different models [7, 8, 11, 6]. To the best of our knowledge, our work is the first (in any setting) to consider verifiable computation for the case where multiple parties provide input. Kamara et al. [18] discuss the case of multi-client verifiable computation in the context of work on server-aided multi-party computation, but leave finding a solution as an open problem.

## 2 Multi-Client Verifiable Computation

We start by introducing the notion of multi-client, non-interactive, verifiable computation ($\mathcal{MVC}$). Let $\kappa$ denote the security parameter. Suppose there are

$n$ clients that wish to evaluate a function $f$ multiple times. Without loss of generality, we assume that each client $P_j$ contributes an $\ell$-bit input and that the output length of $f$ is $\ell$, that is, we have $f : \{0,1\}^{n\ell} \to \{0,1\}^{\ell}$. We abuse notation and let $f$ also denote the representation of the function within some computational model (e.g., as a boolean circuit). Let $x_j^{(i)}$ denote client $P_j$'s input in the $i^{\text{th}}$ execution. For simplicity, we assume that only one client (the first) learns the output; however, we can provide all clients with output by simply running an $\mathcal{MVC}$ scheme in parallel $n$ times (at the cost of increasing the clients' computation by at most a factor of $n$).

**Syntax.** An $n$-party $\mathcal{MVC}$ scheme consists of the following algorithms:

- $(pk_j, sk_j) \leftarrow \mathsf{KeyGen}(1^\kappa, j)$. Each client $P_j$ will run this *key generation algorithm* $\mathsf{KeyGen}$ and obtain a public/private key pair $(pk_j, sk_j)$. Let $\overrightarrow{pk}$ denote the vector $(pk_1, \ldots, pk_n)$ of the public keys of all the clients.
- $(\phi, \xi) \leftarrow \mathsf{EnFunc}(1^\kappa, f)$. The client $P_1$ that is supposed to receive the output will run this *function-encoding algorithm* $\mathsf{EnFunc}$ with a representation of the target function $f$. The algorithm outputs an encoded function $\phi$ and the corresponding decoding secret $\xi$. The encoded function will be sent to the server. The decoding secret is kept private by the client.
- $(\chi_1^{(i)}, \tau^{(i)}) \leftarrow \mathsf{EnInput}_1\big(i, \overrightarrow{pk}, sk_1, \xi, x_1^{(i)}\big)$. When outsourcing the $i^{\text{th}}$ computation to the server, the first client $P_1$ will run this *input-encoding algorithm* $\mathsf{EnInput}_1$ with time period $i$, the public keys $\overrightarrow{pk}$, its secret key $sk_1$, the secret $\xi$ for the encoded function, and its input $x_1^{(i)}$. The output of this algorithm is an encoded input $\chi_1^{(i)}$, which will be sent to the server, and the input decoding secret $\tau^{(i)}$ which will be kept private by the client.
- $\chi_j^{(i)} \leftarrow \mathsf{EnInput}_j\big(i, \overrightarrow{pk}, sk_j, x_j^{(i)}\big)$. When outsourcing the $i^{\text{th}}$ computation to the server, each client $P_j$ (with $j \neq 1$) will run this *input-encoding algorithm* $\mathsf{EnInput}_j$ with time period $i$, the public keys $\overrightarrow{pk}$, its secret key $sk_j$, and its input $x_j^{(i)}$. The output of this algorithm is an encoded input $\chi_j^{(i)}$, which will be sent to the server. We let $\boldsymbol{\chi}^{(i)}$ denote the vector $(\chi_1^{(i)}, \ldots, \chi_n^{(i)})$ of encoded inputs from the clients.
- $\omega^{(i)} \leftarrow \mathsf{Compute}(i, \overrightarrow{pk}, \phi, \boldsymbol{\chi}^{(i)})$. Given the public keys $\overrightarrow{pk}$, the encoded function $\phi$, and the encoded inputs $\boldsymbol{\chi}^{(i)}$, this *computation algorithm* computes an encoded output $\omega^{(i)}$.
- $y^{(i)} \cup \{\perp\} \leftarrow \mathsf{Verify}(i, \xi, \tau^{(i)}, \omega^{(i)})$. The first client $P_1$ runs this *verification algorithm* with the decoding secrets $(\xi, \tau^{(i)})$, and the encoded output $\omega^{(i)}$. The algorithm outputs either a value $y^{(i)}$ (that is supposed to be $f(x_1^{(i)}, \ldots, x_n^{(i)})$), or $\perp$ indicating that the server attempted to cheat.

Of course, to be interesting an $\mathcal{MVC}$ scheme should have the property that the time to encode the input and verify the output is smaller than the time to compute the function from scratch. Correctness of an $\mathcal{MVC}$ scheme can be

defined naturally, that is, the key generation, function encoding, and input encoding algorithms allow the computation algorithm to output an encoded output that will successfully pass the verification algorithm.

### 2.1 Soundness

Intuitively, a verifiable computation scheme is sound if a malicious server cannot convince the honest clients to accept an incorrect output. In our definition, the adversary is given oracle access to generate multiple input encodings.

**Definition 1 (Soundness).** *For a multi-client verifiable-computation scheme $\mathcal{MVC}$, consider the following experiment with respect to an adversarial server $\mathcal{A}$:*

***Experiment*** $\mathbf{Exp}_{\mathcal{A}}^{\text{sound}}[\mathcal{MVC}, f, \kappa, n]$
*For* $j = 1, \ldots, n$*:*
$(pk_j, sk_j) \leftarrow \mathsf{KeyGen}(1^\kappa, j)$,
$(\phi, \xi) \leftarrow \mathsf{EnFunc}(1^\kappa, f)$.
*Initialize counter* $i := 0$
$\omega^* \leftarrow \mathcal{A}^{\mathcal{IN}(\cdot)}(\overrightarrow{pk}, \phi)$;
$y^* \leftarrow \mathsf{Verify}(i, \xi, \tau^{(i)}, \omega^*)$;
*If* $y^* \notin \{\perp, f(x_1^{(i)}, \ldots, x_n^{(i)})\}$,
*output* 1;
*Else output* 0;

***Oracle*** $\mathcal{IN}(x_1, \ldots, x_n)$*:*
$i := i + 1$;
*Record* $(x_1^{(i)}, \ldots, x_n^{(i)}) := (x_1, \ldots, x_n)$.
$(\chi_1^{(i)}, \tau^{(i)}) \leftarrow \mathsf{EnInput}_1\big(i, \overrightarrow{pk}, sk_1, \xi, x_1^{(i)}\big)$
*For* $j = 2, \ldots, n$*:*
$\chi_j^{(i)} \leftarrow \mathsf{EnInput}_j\big(i, \overrightarrow{pk}, sk_j, x_j^{(i)}\big)$.
*Output* $(\chi_1^{(i)}, \ldots, \chi_n^{(i)})$.

*A multi-client verifiable computation scheme $\mathcal{MVC}$ is* sound *if for any $n = \mathsf{poly}(\kappa)$, any function $f$, and any PPT adversary $\mathcal{A}$, there is a negligible function* $\mathsf{negl}$ *such that:*

$$\Pr[\mathbf{Exp}_{\mathcal{A}}^{\text{sound}}[\mathcal{MVC}, f, \kappa, n] = 1] \leq \mathsf{negl}(\kappa).$$

**Selective aborts.** Our $\mathcal{MVC}$ construction described in Section 5 inherits the "selective abort" issue from the single-client scheme of Gennaro et al. [13]; that is, the server may be able to violate soundness if it can send ill-formed responses to the first client and see when that client rejects. In our definition we deal with this issue as in [13] by assuming that the adversary cannot tell when the client rejects. In practice, this issue could be dealt with by having the first client refuse to interact with the server after receiving a single faulty response.

**Adaptive choice of inputs.** As in [13], we define a notion of *adaptive* soundness that allows the adversary to adaptively choose inputs for the clients after seeing the encoded function. (The weaker notion of non-adaptive soundness would require the adversary to fix the clients' inputs in advance, before seeing the encoded function.) Bellare et al. [3] noted that the proof of adaptive soundness in [13] is flawed; it appears to be non-trivial to resolve this issue since it amounts to proving some form of security against selective-opening attacks. Nevertheless, it is reasonable to simply make the assumption that Yao's garbled-circuit construction satisfies the necessary criterion (namely, $\mathsf{aut!}$, as defined in [3]) needed to

prove adaptive security. Similarly, we reduce the adaptive security of our scheme to adaptive authenticity ($\mathsf{aut!}$) of the underlying garbling scheme. Alternately, we can prove non-adaptive soundness based on standard assumptions.

### 2.2 Privacy

We consider two notions of privacy.

**Privacy against the first client.** In our schemes, clients other than the first client clearly do not learn anything about each others' inputs. We define the requirement that the first client not learn anything (beyond the output of the function), either. Namely, given any input vectors $\boldsymbol{x}_0 = (x_1, x_2, \ldots, x_n)$ and $\boldsymbol{x}_1 = (x_1, x_2', \ldots, x_n')$ with $f(x_1, x_2, \ldots, x_n) = f(x_1, x_2', \ldots, x_n')$, the view of the first client when running an execution of the protocol with clients holding inputs $\boldsymbol{x}_0$ should be indistinguishable from the view of the first client when running an execution with clients holding inputs $\boldsymbol{x}_1$.

**Privacy against the server.** Next, we consider *privacy against the server*; that is, the encoded inputs from two distinct inputs should be indistinguishable to the server.

**Definition 2 (Privacy against the server).** *For a multi-client verifiable computation scheme $\mathcal{MVC}$, consider the following experiment with respect to a stateful adversarial server $\mathcal{A}$:*

***Experiment*** $\mathbf{Exp}^{\text{priv}}_{\mathcal{A}}[\mathcal{MVC}, f, \kappa, n, b]$*:*
$(pk_j, sk_j) \leftarrow \mathsf{KeyGen}(1^\kappa, j)$*, for* $j = 1, \ldots, n$.
$(\phi, \xi) \leftarrow \mathsf{EnFunc}(1^\kappa, f)$.
*Initialize counter* $i := 0$
$((x_1^0, \ldots, x_n^0), (x_1^1, \ldots, x_n^1)) \leftarrow \mathcal{A}^{\mathcal{IN}(\cdot)}(\overrightarrow{pk}, \phi)$*;*
*Run* $(\chi_1^{(i)}, \ldots, \chi_n^{(i)}) \leftarrow \mathcal{IN}(x_1^b, \ldots, x_n^b)$*;*
*Output* $\mathcal{A}^{\mathcal{IN}(\cdot)}(\chi_1^{(i)}, \ldots, \chi_n^{(i)})$*;*

*We define the advantage of an adversary $\mathcal{A}$ in the experiment above as:*

$$\mathbf{Adv}^{priv}_{\mathcal{A}}(\mathcal{MVC}, f, \kappa, n) = \left| \begin{array}{r} \Pr[\mathbf{Exp}^{\text{priv}}_{\mathcal{A}}[\mathcal{MVC}, f, \kappa, n, 0] = 1] \\ - \Pr[\mathbf{Exp}^{\text{priv}}_{\mathcal{A}}[\mathcal{MVC}, f, \kappa, n, 1] = 1] \end{array} \right|$$

*$\mathcal{MVC}$ is private against the server if for any $n = \mathsf{poly}(\kappa)$, any function $f$, and any PPT adversary $\mathcal{A}$, there is a negligible function $\mathsf{negl}$ such that:*

$$\mathbf{Adv}^{priv}_{\mathcal{A}}(\mathcal{MVC}, f, \kappa, n) \leq \mathsf{negl}(\kappa).$$

## 3 Building Blocks for $\mathcal{MVC}$

### 3.1 (Projective) Garbling Schemes

Bellare et al. [4] recently formalized a notion of *garbling schemes* that is meant to abstract, e.g., Yao's garbled-circuit protocol [25]. We follow their definition,

since it allows us to abstract the exact properties we need. For completeness, we present their definition below.

A *garbling scheme* [4] is a five-tuple of algorithms $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ with the following properties:

- $y := \mathsf{ev}(f, x)$. Here, $f$ is a bit string that represents a certain function mapping an $\ell$-bit input to an $m$-bit output. (We require that $\ell$ and $m$ be extracted from $f$ in time linear in $|f|$.) For example, $f$ may be a circuit description encoded as detailed in [4]. Hereafter, we abuse the notation and let $f$ also denote the function that $f$ represents. Given the description $f$ and $x \in \{0,1\}^\ell$ as input, $\mathsf{ev}(f, x)$ returns $f(x)$. A garbling scheme is called a *circuit garbling scheme* if $\mathsf{ev} = \mathsf{ev}_{\mathsf{circ}}$ is the canonical circuit-evaluation function.
- $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$. Given the description $f$ as input, $\mathsf{Gb}$ outputs a garbled function $F$ along with an encoding function $e$ and a decoding function $d$.
- $X := \mathsf{En}(e, x)$. Given an encoding function $e$ and $x \in \{0,1\}^\ell$ as input, $\mathsf{En}$ maps $x$ to a *garbled input* $X$. Our scheme will use a *projective garbling scheme*, i.e., the string $e$ encodes a list of *tokens*, one pair for each bit in $x \in \{0,1\}^\ell$. Formally, for all $f \in \{0,1\}^*$, $\kappa \in \mathbb{N}$, $i \in [\ell]$, $x, x' \in \{0,1\}^\ell$ s.t. $x_i = x'_i$, it holds that

$$\Pr\left[\begin{array}{l}(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f),\\ (X_1, \ldots, X_\ell) := \mathsf{En}(e, x), \ : \ X_i = X'_i\\ (X'_1, \ldots, X'_\ell) := \mathsf{En}(e, x')\end{array}\right] = 1.$$

  For a projective garbling scheme $\mathcal{G}$, it is possible to define an additional deterministic algorithm $\mathsf{En}_{\mathrm{proj}}$. Let $(X^0_1, \ldots, X^0_\ell) := \mathsf{En}(e, 0^\ell)$, and $(X^1_1, \ldots, X^1_\ell) := \mathsf{En}(e, 1^\ell)$. The output $\mathsf{En}_{\mathrm{proj}}(e, b, i)$ is defined as $X^b_i$. We refer to $\mathsf{En}_{\mathrm{proj}}$ as the projection algorithm.
- $Y := \mathsf{Ev}(F, X)$. Given a garbled function $F$ and a garbled input $X$ as input, $\mathsf{Ev}$ obtains the garbled output $Y$.
- $y := \mathsf{De}(d, Y)$. Given a decoding function $d$ and a garbled output $Y$, $\mathsf{De}$ maps $Y$ to a final output $y$.

Note that all algorithms except $\mathsf{Gb}$ are deterministic. A garbling scheme must satisfy the following:

1. *Length* condition: $|F|$, $e$, and $d$ depend only on $\kappa$, $\ell$, $m$, and $|f|$.
2. *Correctness* condition: for all $f \in \{0,1\}^*, \kappa \in \mathbb{N}, x \in \{0,1\}^\ell$, it holds that

$$\Pr[(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f) : \ \mathsf{De}(d, \mathsf{Ev}(F, \mathsf{En}(e, x))) = \mathsf{ev}(f, x)] = 1.$$

3. *Non-degeneracy* condition: $e$ and $d$ depends only on $\kappa$, $\ell, m$, $|f|$, and the random coins of $\mathsf{Gb}$.

**Authenticity.** We will employ a garbling scheme that satisfies the *authenticity* property [4]. Loosely speaking, a garbling scheme is authentic if the adversary upon learning a set of tokens corresponding to some input $x$ is unable to produce a set of tokens that correspond to an output different from $f(x)$. Different

notions of authenticity are possible depending on whether the adversary chooses the input $x$ adaptively (i.e., whether it sees the garbled function $F$ before choosing $x$). We adopt the adaptive definition given in [3] because we want our $\mathcal{MVC}$ scheme to achieve adaptive soundness; alternately, we could use a non-adaptive definition of authenticity and achieve non-adaptive soundness.

**Definition 3.** *For a garbling scheme* $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ *consider the following experiment with respect to an adversary* $\mathcal{A}$*.*

***Experiment*** $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{Aut!}_{\mathcal{G}}}[\kappa]$ *:*
$f \leftarrow \mathcal{A}(1^\kappa)$.
$(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$.
$x \leftarrow \mathcal{A}(1^\kappa, F)$.
$X := \mathsf{En}(e, x)$.
$Y \leftarrow \mathcal{A}(1^\kappa, F, X)$.
*If* $\mathsf{De}(d, Y) \neq \perp$ *and* $Y \neq \mathsf{Ev}(F, X)$*, output 1, else 0.*

*A garbling scheme* $\mathcal{G}$ *satisfies the authenticity property if for any PPT adversary* $\mathcal{A}$*, there is a negligible function* $\mathsf{negl}$ *such that*

$$\Pr[\mathbf{Exp}_{\mathcal{A}}^{\mathsf{Aut!}_{\mathcal{G}}}[\kappa] = 1] \leq \mathsf{negl}(\kappa).$$

### 3.2 Fully Homomorphic Encryption

In a (compact) fully-homomorphic encryption scheme $\mathsf{FHE} = (\mathsf{Fgen}, \mathsf{Fenc}, \mathsf{Fdec}, \mathsf{Feval})$, the first three algorithms form a semantically secure public-key encryption scheme. Moreover, $\mathsf{Feval}$ takes a circuit $C$ and a tuple of ciphertexts and outputs a ciphertext that decrypts to the result of applying $C$ to the plaintexts; here, the length of the output ciphertext should be independent of the size of the circuit $C$. We will treat $\mathsf{FHE}$ as a black box.

## 4 Non-Interactive Proxy OT

In this section, we introduce a new primitive called *non-interactive proxy oblivious transfer* ($\mathsf{POT}$), which is a variant and generalization of proxy OT of the notion defined by Naor et al. [22]. In a $\mathsf{POT}$ protocol there are three parties: a sender, a chooser, and a proxy. The *sender* holds input $(x_0, x_1)$, and the *chooser* holds choice bit $b$. At the end of the protocol, the *proxy* learns $x_b$ (but not $x_{1-b}$); the sender and chooser learn nothing. Our definition requires the scheme to be non-interactive, so we omit the term 'non-interactive' from now on.

The generalization we define incorporates public keys for the sender and the chooser, and explicitly takes into account the fact that the protocol may be run repeatedly during multiple time periods. These are needed for the later application to our $\mathcal{MVC}$ construction.

**Syntax.** A proxy OT consists of the following sets of algorithms:

- $(pk_S, sk_S) \leftarrow \mathsf{SetupS}(1^\kappa)$. The sender runs this one-time setup algorithm to generate a public/private key pair $(pk_S, sk_S)$.
- $(pk_C, sk_C) \leftarrow \mathsf{SetupC}(1^\kappa)$. The chooser runs this one-time setup algorithm to generate a public/private key pair $(pk_C, sk_C)$.
- $\alpha \leftarrow \mathsf{Snd}(i, pk_C, sk_S, x_0, x_1)$. In the $i^{\text{th}}$ POT execution the sender, holding input $x_0, x_1 \in \{0,1\}^\kappa$, runs this algorithm to generate a single encoded message $\alpha$ to be sent to the proxy. We refer to $\alpha$ as the sender message.
- $\beta \leftarrow \mathsf{Chs}(i, pk_S, sk_C, b)$. In the $i^{\text{th}}$ POT protocol, the chooser, holding input $b \in \{0,1\}$, runs this algorithm to generate a single encoded message $\beta$ to be sent to the server. We refer to $\beta$ as the chooser message.
- $y := \mathsf{Prx}(i, pk_S, pk_C, \alpha, \beta)$. In the $i^{\text{th}}$ POT protocol, the proxy runs this algorithm using the sender message $\alpha$ and the chooser message $\beta$, and computes the value $y = x_b$.

A proxy OT is correct if the sender algorithm Snd and chooser algorithm Chs produce values that allow the proxy to compute one of two sender inputs based on the chooser's selection bit.

**Sender privacy.** A proxy OT is sender private if the proxy learns only the value of the sender input that corresponds to the chooser's input bit. To serve our purpose, we define sender privacy over multiple executions. We stress that a *single* setup by each party is sufficient to run multiple executions (this is essential for our $\mathcal{MVC}$ construction).

**Definition 4 (Sender Privacy).** *For a proxy OT* $(\mathsf{SetupS}, \mathsf{SetupC}, \mathsf{Snd}, \mathsf{Chs}, \mathsf{Prx})$*, consider the following experiments with respect to an adversarial proxy* $\mathcal{A}$*.*

***Experiment*** $\mathbf{Exp}^{s\text{-}priv}_{\mathcal{A}}[\mathsf{POT}, \kappa, n, e]$*:*
$(pk_S, sk_S) \leftarrow \mathsf{SetupS}(1^\kappa)$.
*For* $j = 1, \ldots, n$*:*
$(pk_{C,j}, sk_{C,j}) \leftarrow \mathsf{SetupC}(1^\kappa)$.
*Output* $\mathcal{A}^{\mathcal{POT}_e(\cdot)}(pk_S, pk_{C,1}, \ldots, pk_{C,n})$.

***Oracle*** $\mathcal{POT}_e(i, j, x_0, x_1, b, x')$*:*
*If a previous query used the same* $(i, j)$ *output* $\perp$ *and terminate.*
*If* $e = 0$*, set* $y_0 := x_0$*,* $y_1 := x_1$*.*
*Else set* $y_b := x_b$*,* $y_{1-b} := x'$*.*
$\alpha \leftarrow \mathsf{Snd}(i, pk_{C,j}, sk_S, y_0, y_1)$.
$\beta \leftarrow \mathsf{Chs}(i, pk_S, sk_{C,j}, b)$.
*Output* $(\alpha, \beta)$.

*Note that the sender messages* $\alpha$ *generated from oracle* $\mathcal{POT}_0$ *(resp.,* $\mathcal{POT}_1$*) would encode the sender's input* $x_b$ *and* $x_{1-b}$ *(resp.,* $x_b$ *and* $x'$*). We define the advantage of an adversary* $\mathcal{A}$ *in the experiment above as:*

$$\mathbf{Adv}^{s\text{-}priv}_{\mathcal{A}}(\mathsf{POT}, \kappa, n) = \left| \begin{array}{l} \Pr[\mathbf{Exp}^{s\text{-}priv}_{\mathcal{A}}[\mathsf{POT}, \kappa, n, 0] = 1] \\ \qquad - \Pr[\mathbf{Exp}^{s\text{-}priv}_{\mathcal{A}}[\mathsf{POT}, \kappa, n, 1] = 1] \end{array} \right|$$

*A proxy OT* $(\mathsf{SetupS}, \mathsf{SetupC}, \mathsf{Snd}, \mathsf{Chs}, \mathsf{Prx})$ *is* sender private*, if for any* $n = \mathsf{poly}(\kappa)$ *and any PPT adversary* $\mathcal{A}$*, there is a negligible function* negl *such that:*

$$\mathbf{Adv}^{s\text{-}priv}_{\mathcal{A}}(\mathsf{POT}, \kappa, n) \leq \mathsf{negl}(\kappa).$$

**Chooser privacy.** A proxy OT is chooser private if the proxy learns no information about the chooser's input bit. To serve our purpose, we define chooser privacy over multiple executions.

**Definition 5 (Chooser Privacy).** *For a proxy OT* $(\mathsf{SetupS}, \mathsf{SetupC}, \mathsf{Snd}, \mathsf{Chs}, \mathsf{Prx})$*, consider the following experiments with respect to an adversarial proxy* $\mathcal{A}$*.*

***Experiment*** $\mathbf{Exp}^{c\text{-}priv}_{\mathcal{A}}[\mathsf{POT}, \kappa, n, e]$*:*
$(pk_S, sk_S) \leftarrow \mathsf{SetupS}(1^\kappa)$.
*For* $j = 1, \ldots, n$*:*
$(pk_{C,j}, sk_{C,j}) \leftarrow \mathsf{SetupC}(1^\kappa)$.
*Output* $\mathcal{A}^{\mathcal{CHS}_e(\cdot)}(pk_S, pk_{C,1}, \ldots, pk_{C,n})$.

***Oracle*** $\mathcal{CHS}_e(i, j, b_0, b_1)$*:*
*If a previous query used the same* $(i, j)$ *output* $\perp$ *and terminate.*
$\beta \leftarrow \mathsf{Chs}(i, pk_S, sk_{C,j}, b_e)$.
*Output* $\beta$.

*We define the advantage of an adversary* $\mathcal{A}$ *in the experiment above as:*

$$\mathbf{Adv}^{c\text{-}priv}_{\mathcal{A}}(\mathsf{POT}, \kappa, n) = \left| \begin{array}{r} \Pr[\mathbf{Exp}^{c\text{-}priv}_{\mathcal{A}}[\mathsf{POT}, \kappa, n, 0] = 1] \\ - \Pr[\mathbf{Exp}^{c\text{-}priv}_{\mathcal{A}}[\mathsf{POT}, \kappa, n, 1] = 1] \end{array} \right|$$

*A proxy OT* $(\mathsf{SetupS}, \mathsf{SetupC}, \mathsf{Snd}, \mathsf{Chs}, \mathsf{Prx})$ *is* chooser private*, if for any* $n = poly(\kappa)$ *and any PPT adversary* $\mathcal{A}$*, there is a negligible function* $\mathsf{negl}$ *such that:*

$$\mathbf{Adv}^{c\text{-}priv}_{\mathcal{A}}(\mathsf{POT}, \kappa, n) \leq \mathsf{negl}(\kappa).$$

### 4.1 Proxy OT from Non-Interactive Key Exchange

**Non-interactive key exchange.** A non-interactive key-exchange (NIKE) protocol allows two parties to generate a shared key based on their respective public keys (and without any direct interaction). That is, let $\mathsf{KEA}_1, \mathsf{KEB}_2$ be the algorithms used by the two parties to generate their public/private keys. $(pk_a, sk_a)$ and $(pk_b, sk_b)$, respectively. Then there are algorithms $\mathsf{KEA}_2$ and $\mathsf{KEB}_2$ such that $\mathsf{KEA}_2(pk_b, sk_a) = \mathsf{KEB}_2(pk_a, sk_b)$. An example is given by static/static Diffie-Hellman key exchange.

Regarding the security of NIKE, to the view of a passive eavesdropper the distribution of the key shared by the two parties should be indistinguishable from a uniform key.

**Definition 6 (Security of NIKE).** *A NIKE* $(\mathsf{KEA}_1, \mathsf{KEA}_2, \mathsf{KEB}_1, \mathsf{KEB}_2)$ *is secure if for any PPT* $\mathcal{A}$*, it holds that* $|p_1 - p_2|$ *is negligible in* $\kappa$*, where*

$$p_1 = \Pr\left[ \begin{array}{l} (pk_a, sk_a) \leftarrow \mathsf{KEA}_1(1^\kappa); \\ (pk_b, sk_b) \leftarrow \mathsf{KEB}_1(1^\kappa) \end{array} : \ \mathcal{A}(pk_a, pk_b, \mathsf{KEA}_2(pk_b, sk_a)) = 1 \right]$$

$$p_2 = \Pr\left[ \begin{array}{l} (pk_a, sk_a) \leftarrow \mathsf{KEA}_1(1^\kappa); \\ (pk_b, sk_b) \leftarrow \mathsf{KEB}_1(1^\kappa); \\ r \leftarrow \{0,1\}^\kappa \end{array} : \ \mathcal{A}(pk_a, pk_b, r) = 1 \right].$$

**Proxy OT from NIKE** We define a protocol for proxy OT below. The main idea is that the sender and the chooser share randomness in the setup stage by

using the key-exchange protocol. Then, using the shared randomness (which is unknown to the proxy), both parties can simply use a one-time pad encryption to transfer their inputs. Let $\mathsf{Prf}$ be a pseudorandom function.

- $(pk_S, sk_S) \leftarrow \mathsf{SetupS}(1^\kappa)$. Run a key exchange protocol on Alice's part, that is, $(pk_a, sk_a) \leftarrow \mathsf{KEA}_1(1^\kappa)$. Set $pk_S := pk_a$ and $sk_S := sk_a$.
- $(pk_C, sk_C) \leftarrow \mathsf{SetupC}(1^\kappa)$. Run a key exchange protocol on Bob's part, that is, $(pk_b, sk_b) \leftarrow \mathsf{KEB}_1(1^\kappa)$. Set $pk_C := pk_b$ and $sk_C := sk_b$.
- $\alpha \leftarrow \mathsf{Snd}(i, pk_C, sk_S, x_0, x_1)$. Let $k$ be the output from the key-exchange protocol, i.e., $k := \mathsf{KEA}_2(pk_C, sk_S)$. Compute $(z_0, z_1, \pi) := \mathsf{Prf}_k(i)$ where $|z_0| = |z_1| = \kappa$ and $\pi \in \{0,1\}$. Then, set $\alpha := (\alpha_0, \alpha_1)$, where

$$\alpha_\pi = z_0 \oplus x_0, \quad \alpha_{1\oplus\pi} = z_1 \oplus x_1.$$

- $\beta \leftarrow \mathsf{Chs}(i, pk_S, sk_C, b)$. Let $k$ be the output from the key exchange protocol, i.e., $k := \mathsf{KEB}_2(pk_S, sk_C)$. Compute $(z_0, z_1, \pi) := \mathsf{Prf}_k(i)$ where $|z_0| = |z_1| = \kappa$ and $\pi \in \{0,1\}$. Then, reveal only the part associated with the choice bit $b$. That is, $\beta := (b \oplus \pi, z_b)$
- $y := \mathsf{Prx}(i, pk_S, pk_C, \alpha, \beta)$. Parse $\alpha$ as $(\alpha_0, \alpha_1)$, and $\beta$ as $(b', z')$. Compute $y := \alpha_{b'} \oplus z'$.

It is easy to see that the scheme satisfies the correctness property. Sender privacy over a single execution easily follows from the fact that the outputs from the key exchange and the pseudorandom function look random. Sender privacy over multiple pairs can also be shown with a hybrid argument. The scheme also hides the choice bit of the chooser from the proxy.

## 5 Construction of $\mathcal{MVC}$

In this section, we present our construction for $\mathcal{MVC}$. Our scheme uses proxy OT to extend the single-client scheme of Gennaro et al. [13] (see Section 1.2 for an overview of the scheme). In the pre-processing stage, the keys for proxy OT are set up, and the first client $P_1$, who will receive the function output, generates a garbled function $F$ and gives it the server. Now delegating computation on input $(x_1, \ldots, x_n)$, where the client $P_j$ holds $x_j \in \{0,1\}^\ell$, is performed as follows:

1. For each $j \in [n]$ and $k \in [\ell]$, do the following in parallel:
   (a) Client $P_1$ computes the following pair for the potential garbled input.

$$X^0_{jk} := \mathsf{En}_{\mathrm{proj}}(e, 0, (j-1)\ell + k), \quad X^1_{jk} := \mathsf{En}_{\mathrm{proj}}(e, 1, (j-1)\ell + k)$$

   (b) A proxy OT protocol is executed in which client $P_1$ plays as the sender with $(X^0_{jk}, X^1_{jk})$ as input, and the client $P_j$ plays as the chooser with the $k^{\text{th}}$ bit of $x_j$ as input; the server plays as the proxy.
2. Using the outputs from the proxy OT protocols, the server evaluates the garbled function $F$ and sends the corresponding garbled output $Y$ to $P_1$.
3. Client $P_1$ decodes $Y$ to obtain the actual output $y$.

Protocol 1

Let $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ be a projective garbling scheme, $\mathsf{FHE} = (\mathsf{Fgen}, \mathsf{Fenc}, \mathsf{Fdec}, \mathsf{Feval})$ be a fully homomorphic encryption scheme, and $(\mathsf{SetupS}, \mathsf{SetupC}, \mathsf{Snd}, \mathsf{Chs}, \mathsf{Prx})$ be a proxy OT scheme. Let $L_j := (j-1)\ell$ and $\mathrm{ID}_{ijk} := (i-1)n\ell + L_j + k$.

- $(pk_j, sk_j) \leftarrow \mathsf{KeyGen}(1^\kappa, j)$. The first client runs the algorithm $\mathsf{SetupS}(1^\kappa)$ to obtain $(pk_1, sk_1)$. For each $2 \le j \le n$, client $P_j$ runs $\mathsf{SetupC}(1^\kappa)$ to generate $(pk_j, sk_j)$.

- $(\phi, \xi) \leftarrow \mathsf{EnFunc}(1^\kappa, f)$. The first client generates $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$, and sets $\phi := F$ and $\xi := (e, d)$.

- $(\chi_1^{(i)}, \tau^{(i)}) \leftarrow \mathsf{EnInput}_1(i, \overrightarrow{pk}, sk_1, \xi, x_1^{(i)})$. Let $a := x_1^{(i)}$ and parse $a$ as $a_1 \dots a_\ell$.
  1. Generate $(\mathrm{PK}_i, \mathrm{SK}_i) \leftarrow \mathsf{Fgen}(1^\kappa)$.
  2. For each $k \in [\ell]$, run $\tilde{X}_{i1k} \leftarrow \mathsf{Fenc}(\mathrm{PK}_i, \mathsf{En}_{\mathrm{proj}}(e, a_k, k))$. Set $\psi_{i1} := (\tilde{X}_{i11}, \dots, \tilde{X}_{i1\ell})$.
  3. For $2 \le j \le n$, do the following:
     (a) For each $k \in [\ell]$, compute
     $$\tilde{X}^0_{ijk} \leftarrow \mathsf{Fenc}(\mathrm{PK}_i, \mathsf{En}_{\mathrm{proj}}(e, 0, L_j + k)),$$
     $$\tilde{X}^1_{ijk} \leftarrow \mathsf{Fenc}(\mathrm{PK}_i, \mathsf{En}_{\mathrm{proj}}(e, 1, L_j + k)),$$
     $$\alpha_{ijk} \leftarrow \mathsf{Snd}(\mathrm{ID}_{ijk}, pk_j, sk_1, \tilde{X}^0_{ijk}, \tilde{X}^1_{ijk}).$$
     (b) Set $\psi_{ij} := (\alpha_{ij1}, \dots, \alpha_{ij\ell})$.
  4. Set $\chi_1^{(i)} := (\mathrm{PK}_i, \psi_{i1}, \dots, \psi_{in})$ and $\tau^{(i)} := \mathrm{SK}_i$.

- $\chi_j^{(i)} \leftarrow \mathsf{EnInput}_j(i, \overrightarrow{pk}, sk_j, x_j^{(i)})$ for $j = 2, \dots n$. Let $a := x_j^{(i)}$ and parse $a$ as $a_1 \dots a_\ell$.
  1. For each $k \in [\ell]$, compute $\beta_{ijk} \leftarrow \mathsf{Chs}(\mathrm{ID}_{ijk}, pk_1, sk_j, a_k)$.
  2. Set $\chi_j^{(i)} := (\beta_{ij1}, \dots, \beta_{ij\ell})$.

- $\omega^{(i)} \leftarrow \mathsf{Compute}(i, \overrightarrow{pk}, \phi, (\chi_1^{(i)}, \dots, \chi_n^{(i)}))$. Parse $\chi_1^{(i)}$ as $(\mathrm{PK}_i, \psi_{i1}, \dots, \psi_{in})$, where $\psi_{i1} = (\tilde{X}_{i11}, \dots, \tilde{X}_{i1\ell})$ and for $2 \le j \le n$, $\psi_{ij} = (\alpha_{ij1}, \dots, \alpha_{ij\ell})$. In addition, for $2 \le j \le n$, parse $\chi_j^{(i)}$ as $(\beta_{ij1}, \dots, \beta_{ij\ell})$. The server does the following:
  1. For $2 \le j \le n$ and for $1 \le k \le \ell$, compute $\tilde{X}_{ijk} := \mathsf{Prx}(\mathrm{ID}_{ijk}, pk_1, pk_j, \alpha_{ijk}, \beta_{ijk})$.
  2. Let $C_F$ denote the circuit representation of $\mathsf{Ev}(F, \cdot)$. Then the server computes $\omega^{(i)} \leftarrow \mathsf{Feval}(\mathrm{PK}_i, C_F, \{\tilde{X}_{ijk}\}_{j \in [n], k \in [\ell]})$.

- $y^{(i)} \cup \{\perp\} \leftarrow \mathsf{Verify}(i, \xi, \tau^{(i)}, \omega^{(i)})$. Parse $\xi$ as $(e, d)$. Client $P_1$ obtains $Y^{(i)} \leftarrow \mathsf{Fdec}(\tau^{(i)}, \omega^{(i)})$, and outputs $y^{(i)} := \mathsf{De}(d, Y^{(i)})$.

**Fig. 1.** A scheme for multi-client non-interactive verifiable computation.

The above scheme allows delegating the computation one-time; intuitively, the sender privacy of the underlying proxy OT makes the server learn the garbled inputs *only for the actual inputs* of all clients. Multiple inputs can be handled using fully-homomorphic encryption as with the single-client case. The detailed protocol (called Protocol 1) is described in Figure 1.

**Correctness and non-triviality.** Correctness of our scheme follows from the correctness of the garbling scheme, the correctness of the fully homomorphic encryption scheme, and the correctness of the proxy OT. Non-triviality of our scheme follows from the fact that (1) the time required (by client $P_1$) for computing $\mathsf{En}_{\mathrm{proj}}$, $\mathsf{Fenc}$, and $\mathsf{Snd}$ is $O(\mathrm{poly}(\kappa)|x^{(i)}|)$, and is independent of the circuit size of $f$, and (2) the time required (by all clients $P_j$ for $2 \leq j \leq n$) for computing $\mathsf{Chs}$ is $O(\mathrm{poly}(\kappa)|x^{(i)}|)$, and is independent of the circuit size of $f$.

**Soundness.** At a high level, the soundness of Protocol 1 follows from the sender security of proxy OT, the semantic security of $\mathsf{FHE}$, and the authenticity property of the garbling scheme $\mathcal{G}$.

**Theorem 1.** *Suppose* $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ *be a projective garbling scheme satisfying the authenticity property,* $\mathsf{FHE}$ *is a semantically secure FHE scheme, and* $(\mathsf{SetupS}, \mathsf{SetupC}, \mathsf{Snd}, \mathsf{Chs}, \mathsf{Prx})$ *is a proxy OT scheme that is sender private. Then Protocol 1 is a sound* $\mathcal{MVC}$ *scheme.*

A proof is given in the following section.

**Privacy.** It is easy to see that Protocol 1 is private against the first client, since the output of $\mathsf{Compute}$ algorithm is basically the encryption of the garbled output. For privacy against the server, we need a proxy OT that hides the chooser's input as well.

**Theorem 2.** *Suppose that* $\mathsf{FHE}$ *is a semantically secure fully homomorphic encryption scheme, and that* $(\mathsf{SetupS}, \mathsf{SetupC}, \mathsf{Snd}, \mathsf{Chs}, \mathsf{Prx})$ *is a proxy OT scheme that is chooser private. Then Protocol 1 is private against the server.*

### 5.1 Proof of Theorem 1

Suppose there exists an adversary $\mathcal{A}$ that breaks the soundness of Protocol 1 with respect to a function $f$.

**Hybrid 0.** Let $p$ be an upper bound on the number of queries $\mathcal{A}$ makes. Consider the following experiment that is slightly different from $\mathbf{Exp}^{\mathrm{sound}}_{\mathcal{A}}[\mathcal{MVC}, f, \kappa, n]$:

**Experiment** $\mathbf{Exp}^{\text{r-sound}}_{\mathcal{A}}[\mathcal{MVC}, f, \kappa, n]$:
$(pk_j, sk_j) \leftarrow \mathsf{KeyGen}(1^\kappa, j)$, for $j = 1, \ldots, n$.
$(\phi, \xi) \leftarrow \mathsf{EnFunc}(1^\kappa, f)$.
Initialize counter $i := 0$
Choose $r \leftarrow [p]$.
$(i^*, \omega^*) \leftarrow \mathcal{A}^{\mathcal{IN}(\cdot)}(\overrightarrow{pk}, \phi)$;

If $i^* \neq r$, output 0 and terminate.
$y^* \leftarrow \mathsf{Verify}(i^*, \xi, \tau^{(i^*)}, \omega^*)$;
If $y^* \neq \perp$ and $y^* \neq f(x_1^{(i^*)}, \ldots, x_n^{(i^*)})$, output 1;
Else output 0;

Since $r$ is chosen uniformly at random, $\mathcal{A}$ would succeed in the above experiment with non-negligble probability (i.e., $\Pr[\mathbf{Exp}_{\mathcal{A}}^{\text{sound}}[\mathcal{MVC}, f, \kappa, n] = 1]/p$).

**Hybrid 1.** In this hybrid, the oracle queries $\mathcal{IN}(x_1^{(i)}, \ldots, x_n^{(i)})$ are handled by the following instead of setting $\chi_1^{(i)} \leftarrow \mathsf{EnInput}_1(i, \overrightarrow{pk}, sk_1, \xi, x_1^{(i)})$

– Run $(\chi_1^{(i)}, \tau^{(i)}) \leftarrow \mathsf{EnInput}_1'(i, \overrightarrow{pk}, sk_1, \xi, (x_1^{(i)}, \ldots, x_n^{(i)}))$.

At a high level, $\mathsf{EnInput}_1'$ is identical to $\mathsf{EnInput}_1$ except it sets the inputs to the sender algorithm $\mathsf{Snd}$ of the proxy OT as follows: For all input bits, $\mathsf{Snd}$ obtains the correct token corresponding to the actual input bit, and a zero string in place of the token corresponding to the other bit. The explicit description of $\mathsf{EnInput}_1'$ is found in Figure 2.

$(\chi_1^{(i)}, \tau^{(i)}) \leftarrow \mathsf{EnInput}_1'(i, \overrightarrow{pk}, sk_1, \xi, (x_1^{(i)}, \ldots, x_n^{(i)}))$

Let $a_j := x_j^{(i)}$ for $j \in [n]$, and parse $a_j$ as $a_{j1} \ldots a_{j\ell}$.

1. Generate $(\mathrm{PK}_i, \mathrm{SK}_i) \leftarrow \mathsf{Fgen}(1^\kappa)$.
2. For each $k \in [\ell]$, run $\tilde{X}_{i1k} \leftarrow \mathsf{Fenc}(\mathrm{PK}_i, \mathsf{En}_{\text{proj}}(e, a_{1k}, k))$. Set $\psi_{i1} := (\tilde{X}_{i11}, \ldots, \tilde{X}_{i1\ell})$.
3. For $2 \leq j \leq n$, do the following:
   (a) For each $k \in [\ell]$, compute

   $b := a_{jk}$, $\tilde{X}_{ijk}^b \leftarrow \mathsf{Fenc}(\mathrm{PK}_i, \mathsf{En}_{\text{proj}}(e, b, L_j + k))$, $\boxed{\tilde{X}_{ijk}^{1-b} := 0^{|X_{ijk}^b|},}$

   $\alpha_{ijk} \leftarrow \mathsf{Snd}(\ell i + L_j + k, pk_j, sk_1, \tilde{X}_{ijk}^0, \tilde{X}_{ijk}^1)$.

   (b) Set $\psi_{ij} := (\alpha_{ij1}, \ldots, \alpha_{ij\ell})$.
4. Set $\chi_1^{(i)} := (\mathrm{PK}_i, \psi_{i1}, \ldots, \psi_{in})$ and $\tau^{(i)} := \mathrm{SK}_i$.

**Fig. 2.** Description of $\mathsf{EnInput}_1'$

It is easy to see that Hybrid 0 and Hybrid 1 are indistinguishable due to the sender privacy of the underlying proxy OT scheme. In the reduction, the adversary breaking sender privacy will simulate experiment $\mathbf{Exp}_{\mathcal{A}}^{\text{r-sound}}[\mathcal{MVC}, f, \kappa, n]$ while invoking the oracle $\mathcal{POT}$ with queries $(i, j, \tilde{X}_{ijk}^0, \tilde{X}_{ijk}^1, x_{jk}^{(i)}, 0^{|\tilde{X}_{ijk}^0|})$ to generate the sender messages of the proxy OT, where $x_{jk}^{(i)}$ is the $k^{\text{th}}$ bit of $x_j^{(i)}$, and for $b \in \{0, 1\}$, the value $\tilde{X}_{ijk}^b$ is the output from $\mathsf{Fenc}(\mathrm{PK}_i, \mathsf{En}_{\text{proj}}(e, b, L_j + k))$.

**Hybrid 2.** In this hybrid, the oracle queries $\mathcal{IN}(x_1^{(i)}, \ldots, x_n^{(i)})$ are handled using the following instead of running $\chi_1^{(i)} \leftarrow \mathsf{EnInput}_1'\left(i, \overrightarrow{pk}, sk_1, \xi, x_1^{(i)}\right)$

1. If $i = r$, run $(\chi_1^{(i)}, \tau^{(i)}) \leftarrow \mathsf{EnInput}_1'(i, \overrightarrow{pk}, sk_1, \xi, (x_1^{(i)}, \ldots, x_n^{(i)}))$;
   Otherwise, $(\chi_1^{(i)}, \tau^{(i)}) \leftarrow \mathsf{EnInput}_1''(i, \overrightarrow{pk}, sk_1, (x_1^{(i)}, \ldots, x_n^{(i)}))$;

At a high level, $\mathsf{EnInput}_1''$ is identical to $\mathsf{EnInput}_1'$ except it replaces the token values to zero strings. The explicit description of $\mathsf{EnInput}_1''$ is found in Figure 3.

$$(\chi_1^{(i)}, \tau^{(i)}) \leftarrow \mathsf{EnInput}_1''(i, \overrightarrow{pk}, sk_1, (x_1^{(i)}, \ldots, x_n^{(i)})$$

Let $a_j := x_j^{(i)}$ for $j \in [n]$, and parse $a_j$ as $a_{j1} \ldots a_{j\ell}$. Let $\lambda$ be the output length of the $\mathsf{En}_{\mathrm{proj}}$ algorithm.

1. Generate $(\mathrm{PK}_i, \mathrm{SK}_i) \leftarrow \mathsf{Fgen}(1^\kappa)$.
2. For each $k \in [\ell]$, compute $\tilde{X}_{i1k} \leftarrow \mathsf{Fenc}(\mathrm{PK}_i, 0^\lambda)$. Set $\psi_{i1} := (\tilde{X}_{i11}, \ldots, \tilde{X}_{i1\ell})$.
3. For $2 \le j \le n$, do the following:
   (a) For each $k \in [\ell]$, compute
   $$b := a_{jk}, \quad \tilde{X}_{ijk}^b \leftarrow \mathsf{Fenc}(\mathrm{PK}_i, 0^\lambda), \quad \tilde{X}_{ijk}^{1-b} := 0^{|X_{ijk}^b|},$$
   $$\alpha_{ijk} \leftarrow \mathsf{Snd}(\ell i + L_j + k, pk_j, sk_1, \tilde{X}_{ijk}^0, \tilde{X}_{ijk}^1).$$
   (b) Set $\psi_{ij} := (\alpha_{ij1}, \ldots, \alpha_{ij\ell})$.
4. Set $\chi_1^{(i)} := (\mathrm{PK}_i, \psi_{i1}, \ldots, \psi_{in})$ and $\tau^{(i)} := \mathrm{SK}_i$.

**Fig. 3.** Description of $\mathsf{EnInput}_1''$

Indistinguishability between Hybrid 1 and Hybrid 2 can be shown with a simple hybrid argument where indistinguishability between two adjacent hybrids holds from the semantic security of the underlying FHE scheme.

**Final Step.** As a final step, we reduce the security to the authenticity of the underlying garbling scheme. In particular, using the adversary $\mathcal{A}$ that succeeds in Hybrid 2 with non-negligible probability, we construct an adversary $\mathcal{B}$ that breaks the authenticity of the underlying garbling scheme. $\mathcal{B}$ works as follows:

$\mathcal{B}$ sends $f$ to the challenger and receives $F$ from it. Then, it simulates Hybrid 2 as follows:

1. Run $(pk_j, sk_j) \leftarrow \mathsf{KeyGen}(1^\kappa, j)$, for $j = 1, \ldots, n$.
2. Let $p$ be the upper bound on the number of queries that $\mathcal{A}$ makes, and choose $r \leftarrow [p]$.
3. Run $(i^*, \omega^*) \leftarrow \mathcal{A}^{\mathcal{IN}(\cdot)}(\overrightarrow{pk}, \phi)$ while handling the query $\mathcal{IN}(x_1^{(i)}, \ldots, x_n^{(i)})$ as follows:

(a) For the input encoding of the first party, if $i = r$, then $\mathcal{B}$ sends $(x_1^{(i)}, \ldots, x_n^{(i)})$ to the challenger and receives the corresponding tokens $(X_1, \ldots, X_{n\ell})$. Using these tokens, $\mathcal{B}$ perfectly simulates $\mathsf{EnInput}'_1(i, \overrightarrow{pk}, sk_1, \xi, (x_1^{(i)}, \ldots, x_n^{(i)}))$ by replacing $\mathsf{En}_{\mathrm{proj}}(e, \cdot, k')$s with $X_{k'}$s. Otherwise, run $\mathsf{EnInput}''_1(i, \overrightarrow{pk}, sk_1, (x_1^{(i)}, \ldots, x_n^{(i)}))$.
(b) For $j = 2, \ldots, n$, run $\chi_j^{(i)} \leftarrow \mathsf{EnInput}\big(i, j, \overrightarrow{pk}, sk_j, x_j^{(i)}\big)$.

4. If $i^* = r$, the adversary $\mathcal{B}$ runs $Y^* \leftarrow \mathsf{Fdec}(\mathrm{SK}_{i^*}, \omega^*)$ and outputs $Y^*$ to the challenger. Otherwise, it outputs $\perp$.

The above simulation is perfect.

We show that $\mathcal{B}$ breaks the authenticity of the underlying garbling scheme with non-negligible probability. Let $\mathsf{Succ}$ be the event that $\mathcal{A}$ succeeds in Hybrid 2, that is, $Y^*$ is a valid encoded output but different from $\mathsf{Ev}(F, X^r)$, where $X^{(r)}$ is the encoded input for $x^{(r)}$. This implies that

$$\Pr[\mathbf{Exp}_{\mathcal{B}}^{\mathsf{Aut!}_{\mathcal{G}}} = 1] \geq \Pr[\mathsf{Succ}].$$

Since by assumption, $\mathcal{G}$ satisfies the authenticity property, we conclude that $\Pr[\mathsf{Succ}]$ must be negligible in $\kappa$, contradiction. This concludes the proof of soundness of Protocol 1.

**Acknowledgments.** We thank the referees for their helpful feedback. This work was supported by the US Army Research Laboratory and the UK Ministry of Defence under Agreement Number W911NF-06-3-0001. The first author was also supported in part by the Intelligence Advanced Research Project Activity (IARPA) via DoI/NBC contract #D11PC20194. The views and conclusions contained here are those of the authors and should not be interpreted as representing the official policies, expressed or implied, of the US Army Research Laboratory, the US Government, IARPA, DoI/NBC, the UK Ministry of Defence, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

## References

1. Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In *37th Intl. Colloquium on Automata, Languages, and Programming (ICALP)*, volume 6198 of *LNCS*, pages 152–163. Springer, 2010.
2. Laszlo Babai. Trading group theory for randomness. In *17th Annual ACM Symposium on Theory of Computing (STOC)*, pages 421–429. ACM Press, 1985.
3. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *Advances in Cryptology — Asiacrypt 2012*, LNCS, pages 134–153. Springer, 2012.
4. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *19th ACM Conf. on Computer and Communications Security (CCS)*, pages 784–796. ACM Press, 2012.

5. Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In *Advances in Cryptology — Crypto 2011*, volume 6841 of *LNCS*, pages 111–131. Springer, 2011.
6. Nir Bitanksy, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. Available at `http://eprint.iacr.org/2012/095`.
7. Ran Canetti, Ben Riva, and Guy Rothblum. Practical delegation of computation using multiple servers. In *18th ACM Conf. on Computer and Communications Security (CCS)*, pages 445–454. ACM Press, 2011.
8. Ran Canetti, Ben Riva, and Guy Rothblum. Two protocols for delegation of computation. In *Information Theoretic Security — ICITS 2012*, volume 7412 of *LNCS*, pages 37–61. Springer, 2012.
9. David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *Advances in Cryptology — Crypto '92*, volume 740 of *LNCS*, pages 89–105. Springer, 1993.
10. Kai-Min Chung, Yael Kalai, and Salil P. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *Advances in Cryptology — Crypto 2010*, volume 6223 of *LNCS*, pages 483–501. Springer, 2010.
11. Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proc. 3rd Innovations in Theoretical Computer Science Conference (ITCS)*, pages 90–112. ACM, 2012.
12. Dario Fiore and Rosario Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *19th ACM Conf. on Computer and Communications Security (CCS)*, pages 501–512. ACM Press, 2012.
13. Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Advances in Cryptology — Crypto 2010*, volume 6223 of *LNCS*, pages 465–482. Springer, 2010.
14. Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. In *40th Annual ACM Symposium on Theory of Computing (STOC)*, pages 113–122. ACM Press, 2008.
15. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.
16. Philippe Golle and Ilya Mironov. Uncheatable distributed computations. In David Naccache, editor, *Cryptographers' Track — RSA 2001*, volume 2020 of *LNCS*, pages 425–440. Springer, April 2001.
17. Susan Hohenberger and Anna Lysyanskaya. How to securely outsource cryptographic computations. In Joe Kilian, editor, *2nd Theory of Cryptography Conference — TCC 2005*, volume 3378 of *LNCS*, pages 264–282. Springer, 2005.
18. Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing multi-party computation. Available at `http://eprint.iacr.org/2011/272`.
19. Joe Kilian. A note on efficient zero-knowledge proofs and arguments. In *24th Annual ACM Symposium on Theory of Computing (STOC)*, pages 723–732. ACM Press, 1992.
20. Joe Kilian. Improved efficient arguments. In Don Coppersmith, editor, *Advances in Cryptology — Crypto '95*, volume 963 of *LNCS*, pages 311–324. Springer, 1995.
21. Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.
22. Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *ACM Conf. Electronic Commerce*, pages 129–139, 1999.

23. Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of correct computation. TCC 2013, to appear. Available at `http://eprint.iacr.org/2011/587`.
24. Bryan Parno, Mariana Raykova, and Vinod Vaikuntanathan. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *9th Theory of Cryptography Conference — TCC 2012*, volume 7194 of *LNCS*, pages 422–439. Springer, 2012.
25. A. C.-C. Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 162–167. IEEE, 1986.