Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers[*]

Rosario Gennaro[†] Craig Gentry[‡] Bryan Parno[§]

February 1, 2010

Abstract

Verifiable Computation enables a computationally weak client to "outsource" the computation of a function F on various inputs $x_1,\ldots,x_k$ to one or more workers. The workers return the result of the function evaluation, e.g., $y_i = F(x_i)$, as well as a proof that the computation of F was carried out correctly on the given value x_i. The verification of the proof should require substantially less computational effort than computing $F(x_i)$ from scratch.

We present a protocol that allows the worker to return a computationally-sound, non-interactive proof that can be verified in $O(m)$ time, where m is the bit-length of the output of F. The protocol requires a one-time pre-processing stage by the client which takes $O(|C|)$ time, where C is the smallest Boolean circuit computing F. Our scheme also provides input and output privacy for the client, meaning that the workers do not learn any information about the x_i or y_i values.

1 Introduction

Several trends are contributing to a growing desire to "outsource" computing from a (relatively) weak computational device to a more powerful computation service. For years, a variety of projects, including SETI@Home [5], Folding@Home [2], and the Mersenne prime search [4], have distributed computations to millions of clients around the Internet to take advantage of their idle cycles. A perennial problem is dishonest clients: end users who modify their client software to return plausible results without performing any actual work [22]. Users commit such fraud, even when the only incentive is to increase their relative ranking on a website listing. Many projects cope with such fraud via redundancy; the same work unit is sent to several clients and the results are compared for consistency. Apart from wasting resources, this provides little defense against colluding users.

A related fear plagues cloud computing, where businesses buy computing time from a service, rather than purchase, provision, and maintain their own computing resources [1, 3]. Sometimes the applications outsourced to the cloud are so critical that it is imperative to rule out accidental errors during the computation. Moreover, in such arrangements, the business providing the computing services may have a strong financial incentive to return incorrect answers, if such answers require less work and are unlikely to be detected by the client.

[*]This research was supported in part by the US Army Research laboratory and the UK Ministry of Defence under Agreement Number W911NF-06-3-0001, as well as by the National Science Foundation (NSF), under award number CCF-0424422. Bryan Parno was supported in part by an NSF Graduate Research Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, U.S. Government, UK Ministry of Defense, UK Government, or NSF. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

[†]IBM T.J.Watson Research Center. rosario@us.ibm.com

[‡]IBM T.J.Watson Research Center. cbgentry@us.ibm.com

[§]CyLab, Carnegie Mellon University. parno@cmu.edu

The proliferation of mobile devices, such as smart phones and netbooks, provides yet another venue in which a computationally weak device would like to be able to outsource a computation, e.g., a cryptographic operation or a photo manipulation, to a third-party and yet obtain a strong assurance that the result returned is correct.

In all of these scenarios, a key requirement is that the amount of work performed by the client to generate and verify work instances must be substantially cheaper than performing the computation on its own. It is also desirable to keep the work performed by the workers as close as possible to the amount of work needed to compute the original function. Otherwise, the worker may be unable to complete the task in a reasonable amount of time, or the cost to the client may become prohibitive.

PRIOR WORK: In the security community, research has focused on solutions based on audits and various forms of secure co-processors. Audit-based solutions [9, 23] typically require the client (or randomly selected workers) to recalculate some portion of the work done by untrusted workers. This may be infeasible for resource-constrained clients and often relies on some fraction of the workers to be honest, or at least non-colluding. Audits based on the time taken to compute the result [25] require detailed knowledge of the hardware employed by the worker.

Secure co-processors [26, 30] provide isolated execution environments, but their tamper-resistance typically makes them quite expensive (thousands of dollars each) and sparsely deployed. The requirements of tamper-resistance also lead to the use of weak CPUs to limit the amount of heat dissipation needed. The growing ubiquity of Trusted Platform Modules (TPMs) [27] in commodity machines promises to improve platform security, but TPMs have achieved widespread deployment in part due to reduced costs (one to five dollars each) that result in little to no physical tamper resistance.

In the cryptographic community, the idea to outsource expensive cryptographic operations to a semi-trusted device has a long history. Chaum and Pedersen define the notion of *wallets with observers* [10], a piece of secure hardware installed by a third party, e.g. a bank, on the client's computer to "help" with expensive computations. The hardware is not trusted by the client who retains assurance that it is performing correctly by analyzing its communication with the bank. Hohenberger and Lysyanskaya formalize this model [16], and present protocols for the computation of modular exponentiations (arguably the most expensive step in public-key cryptography operations). Their protocol requires the client to interact with *two* non-colluding servers. Other work targets specific function classes, such as one-way function inversion [15].

Recent advances in fully-homomorphic encryption [12] allow a worker to compute arbitrary functions over encrypted data, but they do not suffice to provide outsourceable computing. Indeed, fully-homomorphic encryption provides *no guarantee* that the worker performed the correct computation. While our solution does employ fully-homomorphic encryption, we combine it with other techniques to provide verifiability.

The theoretical community has devoted considerable attention to the verifiable computation of arbitrary functions. *Interactive proofs* [6, 14] are a way for a powerful (e.g. super-polynomial) prover to (probabilistically) convince a weak (e.g. polynomial) verifier of the truth of statements that the verifier could not compute on its own. As it is well known, the work on interactive proofs lead to the concept of *probabilistically checkable proofs* (PCPs), where a prover can prepare a proof that the verifier can check in only very few places (in particular only a constant number of bits of the proofs needed for NP languages). Notice, however, that the PCP proof might be very long, potentially too long for the verifier to process. To avoid this complication, Kilian proposed the use of efficient arguments[1] [18, 19] in which the prover sends the verifier a short commitment to the entire proof using a Merkle tree. The prover can then interactively open the bits requested by the verifier (this requires the use of a collision-resistant hash function). A non-interactive solution can be obtained using Micali's CS Proofs [21], which remove interaction from the above argument

[1]We follow the standard terminology: an *argument* is a computationally sound proof, i.e. a protocol in which the prover is assumed to be computationally bounded. In an argument, an infinitely powerful prover can convince the verifier of a false statement, as opposed to a proof where this is information-theoretically impossible or extremely unlikely.

by choosing the bits to open based on the application of a random oracle to the commitment string. In more recent work, which still uses some of the standard PCP machinery, Goldwasser et al. [13] show how to build an interactive proof to verify arbitrary polynomial time computations in almost linear time. They also extend the result to a non-interactive argument for a restricted class of functions.

Therefore, if we restrict our attention to non-interactive protocols, the state of the art offers either Micali's CS Proofs [21] which are arguments that can only be proven in the random oracle model, or the arguments from [13] that can only be used for a restricted class of functions.

OUR CONTRIBUTION. We slightly move away from the notions of proofs and arguments, to define the notion of a *Verifiable Computation Scheme*: this is a protocol between two polynomial time parties, a *client* and a *worker*, to collaborate on the computation of a function $F : \{0,1\}^n \rightarrow \{0,1\}^m$. Our definition uses an amortized notion of complexity for the client: he can perform some expensive pre-processing, but after this stage, he is required to run very efficiently. More specifically, a verifiable computation scheme consists of three phases:

Preprocessing A one-time stage in which the client computes some auxiliary (public and private) information associated with F. This phase can take time comparable to computing the function from scratch, but it is performed only once, and its cost is amortized over all the future executions.

Input Preparation When the client wants the worker to compute $F(x)$, it prepares some auxiliary (public and private) information about x. The public information is sent to the worker.

Output Computation and Verification Once the worker has the public information associated with F and x, it computes a string π_x which encodes the value $F(x)$ and returns it to the client. From the value π_x, the client can compute the value $F(x)$ and verify its correctness.

Notice that this is inherently a non-interactive protocol: the client sends a single message to the worker and vice versa. The crucial efficiency requirement is that Input Preparation and Output Verification must take less time than computing F from scratch (ideally linear time, $O(n+m)$). Also, the Output Computation stage should take roughly the same amount of computation as F.

After formally defining the notion of verifiable computation, we present a verifiable computation scheme for *any* function. Assume that the function F is described by a Boolean circuit C. Then the Preprocessing stage of our protocol takes time $O(|C|)$, i.e., time comparable to computing the function from scratch. Apart from that, the client runs in linear time, as Input Preparation takes $O(n)$ time and Output Verification takes $O(m)$ time. Finally the worker takes time $O(|C|)$ to compute the function for the client.

The computational assumptions underlying the security of our scheme are the security of block ciphers (i.e., the existence of one-way functions) and the existence of a secure fully homomorphic encryption scheme [11,12] (more details below). We stress that our non-interactive protocol works for *any* function (as opposed to Goldwasser et al.'s protocol [13] which works only for a restricted class of functions) and can be proven in the standard model (as opposed to Micali's proofs [21] which require the random oracle model).

Motivation: In our setting, the client must still perform an expensive one-time preprocessing phase. After that, in our scheme, the client runs in linear time. Since the preprocessing stage happens only once, it is important to stress that it can be performed in a trusted environment where the weak client, who does not have the computational power to perform it, outsources it to a trusted party (think of a military application in which the client loads the result of the preprocessing stage performed inside the military base by a trusted server, and then goes off into the field where outsourcing servers may not be trusted anymore – or think of the preprocessing phase executed on the client's home machine and then used by his portable device in the field).

Dynamic and Adaptive Input Choice. We note that in this amortized model of computation, Goldwasser et al.'s protocol [13] can be modified using Kalai and Raz's transformation [17] to achieve a non-interactive scheme (see [24]). However an important feature of our scheme, that is not enjoyed by Goldwasser et al.'s protocol [13], is that the inputs to the computation of F can be chosen in a dynamic and adaptive fashion throughout the execution of the protocol (as opposed to [13] where they must be fixed and known in advance).

Privacy. We also note that our construction has the added benefit of providing input and output privacy for the client, meaning that the worker does not learn any information about x or $F(x)$ (details below). This privacy feature is bundled into the protocol and comes at no additional cost. This is a very important aspect, which should be considered a requirement in real-life applications. After all, if you don't trust the worker to compute the function correctly, why would you trust him with the knowledge of your input data? Homomorphic encryption already solves the problem of computing over private data, but it does not address the problem of efficiently verifying the result. Our work therefore is the first to provide a weak client with the ability to efficiently and verifiably offload computation to an untrusted server in such a way that the input remains secret.

OUR SOLUTION IN A NUTSHELL. Our work is based on the crucial (and somewhat surprising) observation that Yao's Garbled Circuit Construction [28,29], in addition to providing secure two-party computation, also provides a "one-time" verifiable computation. In other words, we can adapt Yao's construction to allow a client to outsource the computation of a function on a single input. More specifically, in the preprocessing stage the client garbles the circuit C according to Yao's construction. Then in the "input preparation" stage, the client reveals the random labels associated with the input bits of x in the garbling. This allows the worker to compute the random labels associated with the output bits, and from them the client will reconstruct $F(x)$. If the output bit labels are sufficiently long and random, the worker will not be able to guess the labels for an incorrect output, and therefore the client is assured that $F(x)$ is the correct output.

Unfortunately, reusing the circuit for a second input x' is insecure, since once the output labels of $F(x)$ are revealed, nothing can stop the worker from presenting those labels as correct for $F(x')$. Creating a new garbled circuit requires as much work as if the client computed the function itself, so on its own, Yao's Circuits do not provide an efficient method for outsourcing computation.

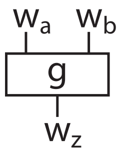
The second crucial idea of the paper is to combine Yao's Garbled Circuit with a fully homomorphic encryption system (e.g., Gentry's recent proposal [12]) to be able to safely reuse the garbled circuit for multiple inputs. More specifically, instead of revealing the labels associated with the bits of input x, the client will encrypt those labels under the public key of a fully homomorphic scheme. A new public key is generated for every input in order to prevent information from one execution from being useful for later executions. The worker can then use the homomorphic property to compute an encryption of the output labels and provide them to the client, who decrypts them and reconstructs $F(x)$.

Since we use the fully-homomorphic encryption scheme in a black-box fashion, we anticipate that any performance improvements in future schemes will directly result in similar performance gains for our protocol as well.

One pre-processing step for many workers: Note that the pre-processing stage is independent of the worker, since it simply produces a Yao-garbled version of the circuit C. Therefore, in addition to being reused many times, this garbled circuit can also be sent to many different workers, which is the usage scenario for applications like Folding@Home [2], which employ a multitude of workers across the Internet.

How to handle malicious workers. In our scheme, if we assume that the worker learns whether or not the client accepts the proof π_x, then for every execution, a malicious worker potentially learns a bit of information about the labels of the Yao-garbled circuit. For example, the worker could try to guess one of the labels, encrypt it with the homomorphic encryption and see if the client accepts. In a sense, the output of the client at the end of the execution can be seen as a very restricted "decryption oracle" for the homomorphic encryption scheme (which is, by definition, not CCA secure). Because of this one-bit leakage, we are unable to prove security in this case.

There are two ways to deal with this. One is to assume that the verification output bit by the client remains private. The other is to repeat the pre-processing stage, i.e. the Yao garbling of the circuit, every



(a)

w_a	w_b	w_z
0	0	g(0,0)
0	1	g(0,1)
1	0	g(1,0)
1	1	g(1,1)

(b)

w_a	w_b	w_z
k_a^0	k_b^0	$k_z^{g(0,0)}$
k_a^0	k_b^1	$k_z^{g(0,1)}$
k_a^1	k_b^0	$k_z^{g(1,0)}$
k_a^1	k_b^1	$k_z^{g(1,1)}$

(c)

w_a	w_b	w_z
k_a^0	k_b^0	$E_{k_a^0}(E_{k_b^0}(k_z^{g(0,0)}))$
k_a^0	k_b^1	$E_{k_a^0}(E_{k_b^1}(k_z^{g(0,1)}))$
k_a^1	k_b^0	$E_{k_a^1}(E_{k_b^0}(k_z^{g(1,0)}))$
k_a^1	k_b^1	$E_{k_a^1}(E_{k_b^1}(k_z^{g(1,1)}))$

(d)

Figure 1: **Yao's Garbled Circuits.** *The original binary gate* **(a)** *can be represented by a standard truth table* **(b)**. *We then replace the 0 and 1 values with the corresponding randomly chosen λ-bit values* **(c)**. *Finally, we use the values for w_a and w_b to encrypt the values for the output wire w_z* **(d)**. *The random permutation of these ciphertexts is the garbled representation of gate g.*

time a verification fails. In this case, in order to preserve a good amortized complexity, we must assume that failures do not happen very often. This is indeed the case in the previous scenario, where the same garbled circuit is used with several workers, under the assumption that only a small fraction of workers will be malicious. Details appear in Section 5.

2 Background

2.1 Yao's Garbled Circuit Construction

We summarize Yao's protocol for two-party private computation [28, 29]. For more details, we refer the interested reader to Lindell and Pinkas' excellent description [20].

We assume two parties, Alice and Bob, wish to compute a function F over their private inputs a and b. For simplicity, we focus on polynomial-time deterministic functions, but the generalization to stochastic functions is straightforward.

At a high-level, Alice converts F into a boolean circuit C. She prepares a garbled version of the circuit, $G(C)$, and sends it to Bob, along with a garbled version, $G(a)$, of her input. Alice and Bob then engage in a series of oblivious transfers so that Bob obtains $G(b)$ without Alice learning anything about b. Bob then applies the garbled circuit to the two garbled outputs to derive a garbled version of the output: $G(F(a,b))$. Alice can then translate this into the actual output and share the result with Bob. Note that this protocol assumes an honest-but-curious adversary model.

In more detail, Alice constructs the garbled version of the circuit as follows. For each wire w in the circuit, Alice chooses two random values $k_w^0, k_w^1 \overset{R}{\leftarrow} \{0,1\}^\lambda$ to represent the bit values of 0 or 1 on that wire. Once she has chosen wire values for every wire in the circuit, Alice constructs a garbled version of each gate g (see Figure 1). Let g be a gate with input wires w_a and w_b, and output wire w_z. Then the garbled version $G(g)$ of g is simply four ciphertexts:

$$\gamma_{00} = E_{k_a^0}(E_{k_b^0}(k_z^{g(0,0)})),\ \gamma_{01} = E_{k_a^0}(E_{k_b^1}(k_z^{g(0,1)})),\ \gamma_{10} = E_{k_a^1}(E_{k_b^0}(k_z^{g(1,0)})),\ \gamma_{11} = E_{k_a^1}(E_{k_b^1}(k_z^{g(1,1)})), \quad (1)$$

where E is an secure symmetric encryption scheme with an "elusive range" (more details below). The order of the ciphertexts is randomly permuted to hide the structure of the circuit (i.e., we shuffle the ciphertexts, so that the first ciphertext does not necessarily encode the output for $(0,0)$).

We refer to w_z^0 and w_z^1 as the "acceptable" outputs for gate g, since they are the only two values that represent valid bit-values for the output wire. Given input keys k_a^x, k_b^y, we will refer to $w_z^{g(x,y)}$ as the "legitimate" output, and $w_z^{1-g(x,y)}$ as the "illegitimate" output.

In Yao's protocol, Alice transfers all of the ciphertexts to Bob, along with the wire values corresponding to the bit-level representation of her input. In other words, she transfers either k_a^0 if her input bit is 0 or k_a^1 if her input bit is 1. Since these are randomly chosen values, Bob learns nothing about Alice's input. Alice and Bob then engage in an oblivious transfer so that Bob can obtain the wire values corresponding to his inputs (e.g., k_b^0 or k_b^1). Bob learns exactly one value for each wire, and Alice learns nothing about his input. Bob can then use the wire values to recursively decrypt the gate ciphertexts, until he arrives at the final output wire values. When he transmits these to Alice, she can map them back to 0 or 1 values and hence obtain the result of the function computation.

2.2 The Security of Yao's Protocol

Lindell and Pinkas prove [20] that Yao is a secure two-party computation protocol under some specific assumptions on the encryption scheme E used to garble the circuit. More specifically the encryption function E needs:

- *Indistinguishable ciphertexts for multiple messages:* For every two vectors of messages $\overline{x}$ and $\overline{y}$, no polynomial time adversary can distinguish between an encryption of $\overline{x}$ and an encryption of $\overline{y}$. Notice that because we require security for multiple messages, we cannot use a one-time pad.
- *An elusive range:* Encryptions under different keys fall into different ranges of the ciphertext space (at least with high probability).
- *An efficiently verifiable range:* Given the key k, it is possible to decide efficiently if a given ciphertext falls into the range of encryptions under k.

We give a formal definition of these properties. Recall that a private encryption scheme is a pair of algorithms (E, D), the encryption and decryption algorithms respectively, that run on input the security parameter λ, a random λ-bit key k, and λ-bit strings (the plaintext and ciphertext, respectively). In the following $\mathtt{negli}()$ denotes a negligible function of its input.

Definition 1 *We say that a private encryption scheme (E, D) is* Yao-secure *if the following properties are satisfied. Assume $k \leftarrow \{0,1\}^\lambda$:*

- Indistinguishability of ciphertexts for multiple messages: *For every efficient adversary A, and every two vectors of ciphertexts $[x_1, \ldots, x_\ell]$ and $[y_1, \ldots, y_\ell]$ (with $\ell = poly(\lambda)$), and $u_i = E_k(x_i)$, $z_i = E_k(y_i)$, we have that*
$$|Prob[A[u_1, \ldots, u_\ell] = 1] - Prob[A[z_1, \ldots, z_\ell] = 1]| < \mathtt{negli}(\lambda)$$
- Elusive Range: *Let* $\mathsf{Range}_\lambda(k) = \{E_k(x)\}_{x \in \{0,1\}^\lambda}$. *For every efficient adversary A we require:*
$$Prob[A(1^\lambda) \in \mathsf{Range}_\lambda(k)] < \mathtt{negli}(\lambda)$$
- Efficiently Verifiable Range: *There exists an efficient machine M such that $M(k, c) = 1$ if and only if $c \in \mathsf{Range}_\lambda(k)$.*

Lindell and Pinkas show [20] that Yao's garbled circuit technique, combined with a secure oblivious transfer protocol, is a secure two-party computation protocol (for semi-honest parties) if E is Yao-secure. They also show how to build Yao-secure encryption schemes based on one-way functions.

2.3 Fully-Homomorphic Encryption

A fully-homomorphic encryption scheme $\mathcal{E}$ is defined by four algorithms: the standard encryption functions **KeyGen**$_\mathcal{E}$, **Encrypt**$_\mathcal{E}$, and **Decrypt**$_\mathcal{E}$, as well as a fourth function **Evaluate**$_\mathcal{E}$. **Evaluate**$_\mathcal{E}$ takes in a circuit C and a tuple of ciphertexts and outputs a ciphertext that decrypts to the result of applying C to the plaintexts. A nontrivial scheme requires that **Encrypt**$_\mathcal{E}$and **Decrypt**$_\mathcal{E}$ operate in time independent of C [11, 12]. More precisely, the time needed to generate a ciphertext for an input wire of C, or decrypt a ciphertext for an output wire, is polynomial in the security parameter of the scheme (independent of C). Note that this implies that the length of the ciphertexts for the output wires is bounded by some polynomial in the security parameter (independent of C).

Gentry recently proposed a scheme, based on ideal lattices, that satisfies these requirements for arbitrary circuits [11, 12]. The complexity of $\mathbf{KeyGen}_{\mathcal{E}}$ in his initial *leveled* fully homomorphic encryption scheme grows linearly with the depth of C. However, under the assumption that his encryption scheme is *circular secure* – i.e., roughly, that it is "safe" to reveal an encryption of a secret key under its associated public key – the complexity of $\mathbf{KeyGen}_{\mathcal{E}}$ is independent of C. See [8, 11, 12] for more discussion on circular-security (and, more generally, key-dependent-message security) as it relates to fully homomorphic encryption.

In this paper, we use fully homomorphic encryption as a black box, and therefore do not discuss the details of any specific scheme.

3 Problem Definition

At a high-level, a verifiable computation scheme is a two-party protocol in which a *client* chooses a function and then provides an encoding of the function and inputs to the function to a *worker*. The worker is expected to evaluate the function on the input and respond with the output. The client then verifies that the output provided by the worker is indeed the output of the function computed on the input provided.

3.1 Basic Requirements

A *verifiable computation scheme* $\mathcal{VC} = (\mathbf{KeyGen}, \mathbf{ProbGen}, \mathbf{Compute}, \mathbf{Verify})$ consists of the four algorithms defined below.

1. $\mathbf{KeyGen}(F, \lambda) \rightarrow (PK, SK)$: Based on the security parameter λ, the randomized *key generation* algorithm generates a public key that encodes the target function F, which is used by the worker to compute F. It also computes a matching secret key, which is kept private by the client.
2. $\mathbf{ProbGen}_{SK}(x) \rightarrow (\sigma_x, \tau_x)$: The *problem generation* algorithm uses the secret key SK to encode the function input x as a public value σ_x which is given to the worker to compute with, and a secret value τ_x which is kept private by the client.
3. $\mathbf{Compute}_{PK}(\sigma_x) \rightarrow \sigma_y$: Using the client's public key and the encoded input, the worker *computes* an encoded version of the function's output $y = F(x)$.
4. $\mathbf{Verify}_{SK}(\tau_x, \sigma_y) \rightarrow y \cup \perp$: Using the secret key SK and the secret "decoding" τ_x, the *verification* algorithm converts the worker's encoded output into the output of the function, e.g., $y = F(x)$ or outputs $\perp$ indicating that σ_y does not represent the valid output of F on x.

A verifiable computation scheme should be both correct and secure. A scheme is correct if the problem generation algorithm produces values that allows an honest worker to compute values that will verify successfully and correspond to the evaluation of F on those inputs. More formally:

Definition 2 (Correctness) *A verifiable computation scheme $\mathcal{VC}$ is* correct *if for any choice of function F, the key generation algorithm produces keys $(PK, SK) \leftarrow \mathbf{KeyGen}(F, \lambda)$ such that, $\forall x \in \mathtt{Domain}(F)$, if $(\sigma_x, \tau_x) \leftarrow \mathbf{ProbGen}_{SK}(x)$ and $\sigma_y \leftarrow \mathbf{Compute}_{PK}(\sigma_x)$ then $y = F(x) \leftarrow \mathbf{Verify}_{SK}(\tau_x, \sigma_y)$.*

Intuitively, a verifiable computation scheme is secure if a malicious worker cannot persuade the verification algorithm to accept an incorrect output. In other words, for a given function F and input x, a malicious worker should not be able to convince the verification algorithm to output $\hat{y}$ such that $F(x) \neq \hat{y}$. Below, we formalize this intuition with an experiment, where $poly(\cdot)$ is a polynomial.

```
Experiment $\mathbf{Exp}_A^{Verif}[\mathcal{VC}, F, \lambda]$
    $(PK, SK) \xleftarrow{R} \mathbf{KeyGen}(F, \lambda)$;
    For $i = 1, \ldots, \ell = poly(\lambda)$;
        $x_i \leftarrow A(PK, x_1, \sigma_1, \ldots, x_i, \sigma_i)$;
        $(\sigma_i, \tau_i) \leftarrow \mathbf{ProbGen}_{SK}(x_i)$;
    $(i, \hat{\sigma}_y) \leftarrow A(PK, x_1, \sigma_1, \ldots, x_\ell, \sigma_\ell)$;
    $\hat{y} \leftarrow \mathbf{Verify}_{SK}(\tau_i, \hat{\sigma}_y)$
    If $\hat{y} \neq \perp$ and $\hat{y} \neq F(x_i)$, output '1', else '0';
```

Essentially, the adversary is given oracle access to generate the encoding of multiple problem instances. The adversary succeeds if it produces an output that convinces the verification algorithm to accept on the wrong output value for a given input value. We can now define the security of the system based on the adversary's success in the above experiment.

Definition 3 (Security) *For a verifiable computation scheme $\mathcal{VC}$, we define the advantage of an adversary A in the experiment above as:*

$$Adv_A^{Verif}(\mathcal{VC}, F, \lambda) = Prob[\mathbf{Exp}_A^{Verif}[\mathcal{VC}, F, \lambda] = 1] \tag{2}$$

A verifiable computation scheme $\mathcal{VC}$ is secure *for a function F, if for any adversary A running in probabilistic polynomial time,*

$$Adv_A^{Verif}(\mathcal{VC}, F, \lambda) \leq \mathtt{negli}(\lambda) \tag{3}$$

where $\mathtt{negli}()$ is a negligible function of its input.

In the above definition, we could have also allowed the adversary to select the function F. However, our protocol is a verifiable computation scheme that is secure for *all* F, so the above definition suffices.

3.2 Input and Output Privacy

While the basic definition of a verifiable computation protects the integrity of the computation, it is also desirable that the scheme protect the secrecy of the input given to the worker(s). We define input privacy based on a typical indistinguishability argument that guarantees that *no* information about the inputs is leaked. Input privacy, of course, immediately yields output privacy.

Intuitively, a verifiable computation scheme is *private* when the public outputs of the problem generation algorithm **ProbGen** over two different inputs are indistinguishable; i.e., nobody can decide which encoding is the correct one for a given input. More formally consider the following experiment: the adversary is given the public key for the scheme and selects two inputs x_0, x_1. He is then given the encoding of a randomly selected one of the two inputs and must guess which one was encoded. During this process the adversary is allowed to request the encoding of any input he desires. The experiment is described below. The oracle $\mathbf{PubProbGen}_{SK}(x)$ calls $\mathbf{ProbGen}_{SK}(x)$ to obtain (σ_x, τ_x) and returns only the public part σ_x.

```
Experiment $\mathbf{Exp}_A^{Priv}[\mathcal{VC}, F, \lambda]$
    $(PK, SK) \xleftarrow{R} \mathbf{KeyGen}(F, \lambda)$;
    $(x_0, x_1) \leftarrow A^{\mathbf{PubProbGen}_{SK}(\cdot)}(PK)$
    $(\sigma_0, \tau_0) \leftarrow \mathbf{ProbGen}_{SK}(x_0)$;
    $(\sigma_1, \tau_1) \leftarrow \mathbf{ProbGen}_{SK}(x_1)$;
    $b \xleftarrow{R} \{0, 1\}$;
    $\hat{b} \leftarrow A^{\mathbf{PubProbGen}_{SK}(\cdot)}(PK, x_0, x_1, \sigma_b)$
    If $\hat{b} = b$, output '1', else '0';
```

Definition 4 (Privacy) *For a verifiable computation scheme $\mathcal{VC}$, we define the advantage of an adversary A in the experiment above as:*

$$Adv_A^{Priv}(\mathcal{VC}, F, \lambda) = Prob[\mathbf{Exp}_A^{Priv}[\mathcal{VC}, F, \lambda] = 1] \tag{4}$$

A verifiable computation scheme $\mathcal{VC}$ is private *for a function F, if for any adversary A running in probabilistic polynomial time,*

$$Adv_A^{Priv}(\mathcal{VC}, F, \lambda) \leq \mathtt{negli}(\lambda) \tag{5}$$

where $\mathtt{negli}()$ is a negligible function of its input.

An immediate consequence of the above definition is that in a private scheme, the encoding of the input must be probabilistic (since the adversary can always query x_0, x_1 to the **PubProbGen** oracle, and if the answer were deterministic, he could decide which input is encoded in σ_b).

A similar definition can be made for output privacy.

3.3 Efficiency

The final condition we require from a verifiable computation scheme is that the time to encode the input and verify the output must be smaller than the time to compute the function from scratch.

Definition 5 (Outsourceable) *A $\mathcal{VC}$ can be outsourced if it permits efficient generation and efficient verification. This implies that for any x and any σ_y, the time required for* $\mathbf{ProbGen}_{SK}(x)$ *plus the time required for* $\mathbf{Verify}(\sigma_y)$ *is $o(T)$, where T is the time required to compute $F(x)$.*

Some functions are naturally outsourceable (i.e., they can be outsourced with no additional mechanisms), but many are not.

Notice that we are not including the time to compute the key generation algorithm (i.e., the encoding of the function itself). Therefore, the above definition captures the idea of an outsourceable verifiable computation scheme which is more efficient than computing the function in an *amortized* sense, since the cost of encoding the function can be amortized over many input computations.

4 An Efficient Verifiable-Computation Scheme with Input/Output Privacy

4.1 Protocol Definition

We are now ready to describe our scheme. Informally, our protocol works as follows. The key generation algorithm consists of running Yao's garbling procedure over a Boolean circuit computing the function F: the public key is the collection of ciphertexts representing the garbled circuit, and the secret key consists of all the random wire labels. The input is encoded in two steps: first a fresh public/secret key pair for a homomorphic encryption scheme is generated, and then the labels of the correct input wires are encrypted with it. These ciphertexts constitute the public encoding of the input, while the secret key is kept private by the client. Using the homomorphic properties of the encryption scheme, the worker performs the computation steps of Yao's protocol, but working over ciphertexts (i.e., for every gate, given the encrypted labels for the correct input wires, obtain an encryption of the correct output wire, by applying the homomorphic encryption over the circuit that computes the "double decryption" in Yao's protocol). At the end, the worker will hold the encryption of the labels of the correct output wires. He returns these ciphertexts to the client who decrypts them and then computes the output from them. We give a detailed description below.

Protocol $\mathcal{VC}$.

1. **KeyGen**$(F,\lambda) \rightarrow (PK,SK)$: Represent F as a circuit C. Following Yao's Circuit Construction (see Section 2.1), choose two values, $w_i^0, w_i^1 \overset{R}{\leftarrow} \{0,1\}^\lambda$ for each wire w_i. For each gate g, compute the four ciphertexts $(\gamma_{00}^g, \gamma_{01}^g, \gamma_{10}^g, \gamma_{11}^g)$ described in Equation 1. The public key PK will be the full set of ciphertexts, i.e., $PK \leftarrow \cup_g(\gamma_{00}^g, \gamma_{01}^g, \gamma_{10}^g, \gamma_{11}^g)$, while the secret key will be the wire values chosen: $SK \leftarrow \cup_i(w_i^0, w_i^1)$.
2. **ProbGen**$_{SK}(x) \rightarrow \sigma_x$: Run the doubly-homomorphic encryption scheme's key generation algorithm to create a new key pair: $(PK_{\mathcal{E}}, SK_{\mathcal{E}}) \leftarrow$ **KeyGen**$_{\mathcal{E}}(\lambda)$. Let $w_i \subset SK$ be the wire values representing the binary expression of x. Set $\sigma_x \leftarrow (PK_\varepsilon, \textbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, w_i))$ and $\tau_x \leftarrow SK_{\mathcal{E}}$.
3. **Compute**$_{PK}(\sigma_x) \rightarrow \sigma_y$: Calculate **Encrypt**$_{\mathcal{E}}(PK_{\mathcal{E}}, \gamma_i)$. Construct a circuit Δ that on input w, w', γ outputs $D_w(D_{w'}(\gamma))$, where D is the decryption algorithm corresponding to the encryption E used in Yao's garbling (therefore Δ computes the appropriate decryption in Yao's construction). Calculate **Evaluate**$_{\mathcal{E}}(\Delta$, **Encrypt**$_{\mathcal{E}}(PK_{\mathcal{E}}, w_i)$, **Encrypt**$_{\mathcal{E}}(PK_{\mathcal{E}}, \gamma_i))$ repeatedly, to decrypt your way through the ciphertexts, just as in the evaluation of Yao's garbled circuit. The result is $\sigma_y \leftarrow \textbf{Encrypt}_{\mathcal{E}}(PK_\varepsilon, \bar{w_i})$, where $\bar{w_i}$ are the wire values representing $y = F(x)$ in binary.
4. **Verify**$_{SK}(\sigma_y) \rightarrow y \cup \perp$: Use $SK_{\mathcal{E}}$ to decrypt **Encrypt**$_{\mathcal{E}}(PK_\varepsilon, \bar{w_i})$, obtaining $\bar{w_i}$. Use SK to map the wire values to an output y. If the decryption or mapping fails, output $\perp$.

Remark: *On verifying ciphertext ranges in an encrypted form.* Recall that Yao's scheme requires the encryption scheme E to have an *efficiently verifiable range*: Given the key k, it is possible to decide efficiently if a given ciphertext falls into the range of encryptions under k. In other words, there exists an efficient machine M such that $M(k,\gamma) = 1$ if and only if $\gamma \in \mathsf{Range}_\lambda(k)$. This is necessary to "recognize" which ciphertext to pick among the four ciphertexts associated with each gate.

In our verifiable computation scheme $\mathcal{VC}$, we need to perform this check using an encrypted form of the key $c = \textbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, k)$. When applying the homomorphic properties of $\mathcal{E}$ to the range testing machine M, the worker obtains an encryption of 1 for the correct ciphertext, and an encryption of 0 for the others. Of course he is not able to distinguish which one is the correct one.

The worker then proceeds as follows: for the four ciphertexts $\gamma_1, \gamma_2, \gamma_3, \gamma_4$ associated with a gate g, he first computes $c_i = \textbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, M(k,\gamma_i))$ using the homomorphic properties of $\mathcal{E}$ over the circuit describing M. Note that only one of these ciphertexts encrypts a 1, exactly the one corresponding to the correct γ_i. Then the worker computes $d_i = \textbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, D_k(\gamma_i))$ using the homomorphic properties of $\mathcal{E}$ over the decryption circuit Δ. Note that $k' = \Sigma_i M(k,\gamma_i) D_k(\gamma_i)$ is the correct label for the output wire. Therefore, the worker can use the homomorphic properties of $\mathcal{E}$ to compute $c = \textbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, k') = \textbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, \Sigma_i M(k,\gamma_i) D_k(\gamma_i))$ from c_i, d_i, as desired.

4.2 Proof of Security

The main result of our paper is the following.

Theorem 1 *Let E be a Yao-secure symmetric encryption scheme and $\mathcal{E}$ be a semantically secure homomorphic encryption scheme. Then protocol $\mathcal{VC}$ is a secure, outsourceable and private verifiable computation scheme.*

The proof of Theorem 1 requires two high-level steps. First, we show that Yao's garbled circuit scheme is a one-time secure verifiable computation scheme, i.e. a scheme that can be used to compute F securely on one input. Then, by using the semantic security of the homomorphic encryption scheme, we reduce the security of our scheme (with multiple executions) to the security of a single execution where we expect the adversary to cheat.

4.3 Proof Sketch of Yao's Security for One Execution

Consider the verifiable computation scheme $\mathcal{VC}_{Yao}$ defined as follows:

Protocol $\mathcal{VC}_{Yao}$.

1. **KeyGen**$(F,\lambda) \rightarrow (PK,SK)$: Represent F as a circuit C. Following Yao's Circuit Construction (see Section 2.1), choose two values, $w_i^0, w_i^1 \overset{R}{\leftarrow} \{0,1\}^\lambda$ for each wire w_i. For each gate g, compute the four ciphertexts $(\gamma_{00}^g, \gamma_{01}^g, \gamma_{10}^g, \gamma_{11}^g)$ described in Equation 1. The public key PK will be the full set of ciphertexts, i.e, $PK \leftarrow \cup_g(\gamma_{00}^g, \gamma_{01}^g, \gamma_{10}^g, \gamma_{11}^g)$, while the secret key will be the wire values chosen: $SK \leftarrow \cup_i(w_i^0, w_i^1)$.
2. **ProbGen**$_{SK}(x) \rightarrow \sigma_x$: Reveal the labels of the input wires associated with x. In other words, let $w_i \subset SK$ be the wire values representing the binary expression of x, and set $\sigma_x \leftarrow (PK_\varepsilon, w_i)$. τ_x is the empty string.
3. **Compute**$_{PK}(\sigma_x) \rightarrow \sigma_y$: Compute the decryptions in Yao's protocol to obtain the labels of the correct output wires. Set σ_y to be these labels.
4. **Verify**$_{SK}(\sigma_y) \rightarrow y \cup \perp$: Use SK to map the wire values in σ_y to the binary representation of the output y. If the mapping fails, output $\perp$.

Theorem 2 *$\mathcal{VC}_{Yao}$ is a correct verifiable computation scheme.*

Proof of Theorem 2: The proof of correctness follows directly from the proof of correctness for Yao's garbled circuit construction [20]. Using C and $\tilde{x}$ will produce a $\tilde{y}$ that represents the correct evaluation of $F(x)$. ■

We prove that $\mathcal{VC}_{Yao}$ is a *one-time* secure verifiable computation scheme. The definition of *one-time secure* is the same as Definition 3 except that in experiment $\mathbf{Exp}_A^{Verif}$, the adversary is allowed to query the oracle **ProbGen**$_{SK}(\cdot)$ only once (i.e., $\ell = 1$) and must cheat on that input.

Intuitively, an adversary who violates the security of this scheme must either guess the "incorrect" random value $k_w^{1-y_i}$ for one of the output bit values representing y, or he must break the encryption scheme used to encode the "incorrect" wire values in the circuit. The former happens with probability $\leq \frac{1}{2^\lambda}$, i.e., negligible in λ. The latter violates our security assumptions about the encryption scheme. We formalize this intuition below using an hybrid argument similar to the one used in [20].

Theorem 3 *Let E be a Yao-secure symmetric encryption scheme. Then $\mathcal{VC}_{Yao}$ is a one-time secure verifiable computation scheme.*

Proof of Theorem 3: Assume w.l.o.g. that the function F outputs a single bit (at the end of the proof we show how to deal with the case of multiple-bit outputs). Assume a canonical order on the gates in the circuit computing F, and let m be the number of such gates. Let PK be the garbled circuit obtained by running **KeyGen**(F,λ).

Fix any adversary A; we show that for A, the probability of successfully cheating is negligible in λ, if the encryption scheme E is Yao-secure. We do this by defining a series of *hybrid* experiments where we change the setting in which A is run, but in a controlled way: each experiment in the series will be *computationally indistinguishable* from the previous one, if the security of the encryption scheme holds. The first experiment in the series is $\mathbf{Exp}_A^{Verif}$. In the last experiment, we will show that information-theoretically A can cheat only with negligible probability, therefore proving that in order to cheat in the original experiment, A must distinguish between two experiments in the series, and thus break the encryption scheme.

We denote with $H_A^i[\mathcal{VC},F,\lambda]$, the i^{th} hybrid experiment, run with an adversary A, verifiable computation scheme $\mathcal{VC}$, function F and security parameter λ. All experiments output a Boolean value, and therefore we can define $Adv_A^i(\mathcal{VC},F,\lambda) = Prob[H_A^i[\mathcal{VC},F,\lambda] = 1]$.

Define

$$p_b = Prob[A \text{ in } \mathbf{Exp}_A^{Verif}[\mathcal{VC}_{Yao},F,\lambda] \text{ outputs } x \text{ s.t. } F(x) = b]$$

Note that we can estimate these probabilities by running the experiment many times. Set β to be the bit such that $p_\beta \geq p_{\bar{\beta}}$. Notice that $p_\beta \geq 1/2$.

Experiment $H^0_A[\mathcal{VC}_{Yao}, F, \lambda]$ **:** This experiment is exactly like $\mathbf{Exp}^{Verif}_A[\mathcal{VC}_{Yao}, F, \lambda]$ except that when A queries **ProbGen** on the input x (recall that we are considering the case where the adversary only submits a single input value and must cheat on that input), the oracle selects a random[2] x' such that $F(x') = \beta$ and returns $\sigma_{x'}$, where $(\sigma_{x'}, \tau_{x'}) \leftarrow \mathbf{ProbGen}_{SK}(\underline{x'})$. The experiment's output bit is set to 1 if A manages to cheat over input x', i.e. produces a valid proof for β (and to 0 otherwise).

Lemma 1 *If E is a Yao-secure encryption scheme, then for all efficient adversaries A we have $|Adv^0_A(\mathcal{VC}_{Yao}, F, \lambda) - Adv^{Verif}_A(\mathcal{VC}_{Yao}, F, \lambda)| \leq \mathtt{negli}(\lambda)$.*

Proof of Lemma 1: The Lemma follows from the security of Yao's two-party computation protocol [20].

Recall that in Yao's protocol, two parties P_1 and P_2 want to compute a function F over inputs x and y privately held respectively by P_1 and P_2, without revealing any information about their inputs except the value $F(x,y)$. The protocol goes as follows: P_1 garbles a circuit computing the function F, and gives to P_2 the labels of his input x. Moreover, P_1 and P_2 engage in OT protocols to give P_2 the labels of her input y, without revealing this input to P_1. Then P_2 executes the circuit on his own and sends the output label to P_1, who reveals the output of the function $F(x,y)$. Note that P_1 sends his input labels in the clear to P_2. The intuition is that P_1's input remains private since P_2 can't associate the labels with the bit values they represent. This intuition is formalized in the proof in [20].

Therefore we reduce the indistinguishability of $H^0_A[\mathcal{VC}_{Yao}, F, \lambda]$ and $\mathbf{Exp}^{Verif}_{A^*}[\mathcal{VC}_{Yao}, F, \lambda]$ to the security of Yao's protocol. We show that if there exists A such that

$$|Adv^0_A(\mathcal{VC}_{Yao}, F, \lambda) - Adv^{Verif}_A(\mathcal{VC}_{Yao}, F, \lambda)| > \varepsilon$$

with non-negligible ε, then we can learn some information about P_1's input with roughly the same advantage.

Suppose we run Yao's two-party protocol between P_1 and P_2 with the function F computed over just P_1's input x'. We assume that P_1's input is chosen with the right distribution[3] (i.e. $F(x') = \beta$). For any two values x, x', with $F(x) = F(x')$, the security of Yao's protocol implies that no efficient player P_2 can distinguish if x or x' was used.

We build a simulator S that plays the role of P_2 and distinguishes between the two input cases, with probability $p_\beta \varepsilon$, thus creating a contradiction.

The protocol starts with P_1 sending the garbled circuit PK and the encoding of his input $\sigma_{x'}$. The simulator computes the label ℓ associated with the output $F(x')$. At this point the simulator engages A over the input PK, and A requests the encoding of an input x. If $F(x) \neq \beta$ the simulator tosses a random coin, and outputs the resulting bit. Notice however that with probability p_β, $F(x) = \beta = F(x')$. In this case, the simulator provides A with the encoding $\sigma_{x'}$, and returns as its output the experiment bit.

Notice that if $x = x'$ we are running $\mathbf{Exp}^{Verif}_A[\mathcal{VC}_{Yao}, F, \lambda]$, while if $x \neq x'$ we are running $H^0_{A^*}[\mathcal{VC}_{Yao}, F, \lambda]$. Therefore the simulator distinguishes between the two input values exactly with probability $p_\beta \varepsilon$, therefore creating a contradiction. ■

Experiment $H^i_A[\mathcal{VC}_{Yao}, F, \lambda]$ **for** $i = 1, \ldots, m$**:** During the i^{th} experiment the **ProbGen** oracle still chooses a random value x' to answer A's query as in $H^0_A[\mathcal{VC}_{Yao}, F, \lambda]$. This value x' defines $0/1$ values for all the wires in the circuit. We say that a label w^b for wire w is *active* if the value of wire w when the circuit is computed over x' is b. We now define a family of fake garbled circuits PK^i_{fake} for $i = 0, \ldots, m$, as follows. For gate g_j with $j \leq i$, if w^b is the active label associated with its output wire w, then *all* four ciphertexts associated with g_j encrypt w^b. For gate g_j, with $j > i$, the four ciphertexts are computed correctly as in Yao's garbling technique, where the value encrypted depends on the keys used to encrypt it. Notice that $PK^0_{fake} = PK$ since for all of the gates, the ciphertexts are computed correctly. The experiment's output bit is still set to 1 if A manages to cheat over input x', i.e. produces a valid proof for β (and to 0 otherwise).

[2]Since F is a Boolean function, w.l.o.g. we can assume that we can efficiently sample x' such that $F(x') = b$.

[3]We can assume this since the security of Yao's protocol is for all inputs, so in particular for this distribution.

Lemma 2 *If E is a Yao-secure encryption scheme, then for all efficient adversaries A we have $|Adv^i_A(\mathcal{VC}_{Yao}, F, \lambda) - Adv^{i-1}_A(\mathcal{VC}_{Yao}, F, \lambda)| \leq \mathtt{negli}(\lambda)$.*

This lemma is actually proven in [20], and we refer the reader to it for a full proof. Intuitively, the lemma follows from the ciphertext indistinguishability of the encryption scheme E.

Lemma 3 $Adv^m_A(\mathcal{VC}_{Yao}, F, \lambda) = 2^{-\lambda}$

Proof of Lemma 3: Recall that $Adv^m_A(\mathcal{VC}_{Yao}, F, \lambda)$ is the probability that A manages to cheat over input x', i.e., to provide the incorrect output label. However, the view of A is information-theoretically independent of that label, since the incorrect output label is inactive and has not been encrypted in the garbled circuit PK^m_{fake}. Since labels are chosen as random λ-bit strings, the probability of guessing the incorrect output label is exactly $2^{-\lambda}$. ■

This completes the proof of Theorem 3. ■

Remark: This proof does not readily extend to the case of a function F with multiple output bits, because in that case it might not be possible to sample an x which produces a specific output (think of a one-way function F for example). However, notice that if the output is n bits, then the value y computed by a successful cheating adversary must be different from $F(x)$ in at least one bit. Thus, at the beginning of the simulation, we can try to guess the bit on which the adversary will cheat and then run the proof for the 1-bit case. Our guess will be right with probability $1/n$.

4.4 Proof of Theorem 1

The proof of Theorem 1 follows from Theorem 2 and the semantic security of the homomorphic encryption scheme. More precisely, we show that if the homomorphic encryption scheme is semantically secure, then we can transform (via a simulation) a successful adversary against the full verifiable computation scheme $\mathcal{VC}$ into an attacker for the one-time secure protocol $\mathcal{VC}_{Yao}$. The intuition is that for each query, the labels in the circuit are encrypted with a semantically-secure encryption scheme (the homomorphic scheme), so multiple queries do not help the adversary to learn about the labels, and hence if he cheats, he must be able to cheat in the one-time case as well.

Proof of Theorem 1: Let us assume for the sake of contradiction that there is an adversary A such that $Adv^{Verif}_A(\mathcal{VC}, F, \lambda) \geq \varepsilon$, where ε is non-negligible in λ. We use A to build another adversary A' which queries the **ProbGen** oracle only once, and for which $Adv^{Verif}_{A'}(\mathcal{VC}_{Yao}, F, \lambda) \geq \varepsilon'$, where ε' is close to ε. The details of A' follow.

A' receives as input the garbled circuit PK. It activates A with the same input. Let ℓ be an upper bound on the number of queries that A makes to its **ProbGen** oracle. The adversary A' chooses an index i at random between 1 and ℓ and continues as follows. For the j^{th} query by A, with $j \neq i$, A' will respond by (i) choosing a random private/public key pair for the homomorphic encryption scheme $(PK^j_{\mathcal{E}}, SK^j_{\mathcal{E}})$ and (ii) encrypting random λ-bit strings under $PK^j_{\mathcal{E}}$. For the i^{th} query, x, the adversary A' gives x to its own **ProbGen** oracle and receives σ_x, the collection of active input labels corresponding to x. It then generates a random private/public key pair for the homomorphic encryption scheme $(PK^i_{\mathcal{E}}, SK^i_{\mathcal{E}})$, and it encrypts σ_x (label by label) under $PK^i_{\mathcal{E}}$.

Once we prove the Lemma 4 below, we have our contradiction and the proof of Theorem 1 is complete ■

Lemma 4 $Adv^{Verif}_{A'}(\mathcal{VC}_{Yao}, F, \lambda) \geq \varepsilon'$ *where ε' is non-negligible in λ.*

Proof of Lemma 4: This proof also proceeds by defining, for any adversary A, a set of hybrid experiments $\mathcal{H}^k_A(\mathcal{VC}, F, \lambda)$ for $k = 0, \ldots, \ell - 1$. We define the experiments below. Let i be an index randomly selected between 1 and ℓ as in the proof above.

Experiment $\mathcal{H}_A^k(\mathcal{VC},F,\lambda)=1]$: In this experiment, we change the way the oracle **ProbGen** computes its answers. For the j^{th} query:

- $j \leq k$ *and* $j \neq i$: The oracle will respond by (i) choosing a random private/public key pair for the homomorphic encryption scheme $(PK_{\mathcal{E}}^j, SK_{\mathcal{E}}^j)$ and (ii) encrypting random λ-bit strings under $PK_{\mathcal{E}}^j$.
- $j > k$ *or* $j = i$: The oracle will respond exactly as in $\mathcal{VC}$, i.e. by (i) choosing a random private/public key pair for the homomorphic encryption scheme $(PK_{\mathcal{E}}^j, SK_{\mathcal{E}}^j)$ and (ii) encrypting the correct input labels in Yao's garbled circuit under $PK_{\mathcal{E}}^j$.

In the end, the bit output by the experiment $\mathcal{H}_A^k$ is 1 if A successfully cheats on the i^{th} input and otherwise is 0. We denote with $Adv_A^k(\mathcal{VC},F,\lambda) = Prob[\mathcal{H}_A^k(\mathcal{VC},F,\lambda)=1]$. Note that

- $\mathcal{H}_A^0(\mathcal{VC},F,\lambda)$ is identical to the experiment $\mathbf{Exp}_A^{Verif}[\mathcal{VC},F,\lambda]$, except for the way the bit is computed at the end. Since the index i is selected at random between 1 and ℓ, we have that

$$Adv_A^0(\mathcal{VC},F,\lambda) = \frac{Adv_A^{Verif}(\mathcal{VC},F,\lambda)}{\ell} \geq \frac{\varepsilon}{\ell}$$

- $\mathcal{H}_A^{\ell-1}(\mathcal{VC},F,\lambda)$ is equal to the simulation conducted by A' above, so

$$Adv_A^{\ell-1}(\mathcal{VC},F,\lambda) = Adv_{A'}^{Verif}(\mathcal{VC}_{Yao},F,\lambda)$$

If we prove for $k=0,\ldots,\ell-1$ that experiments $\mathcal{H}_A^k(\mathcal{VC},F,\lambda)$ and $\mathcal{H}_A^{k-1}(\mathcal{VC},F,\lambda)$ are computationally indistinguishable, that is for every A

$$|Adv_A^k(\mathcal{VC},F,\lambda) - Adv_A^{k-1}(\mathcal{VC},F,\lambda)| \leq \texttt{negli}(\lambda) \tag{6}$$

we are done, since that implies that

$$Adv_{A'}^{Verif}(\mathcal{VC}_{Yao},F,\lambda) \geq \frac{\varepsilon}{\ell} - \ell \cdot \texttt{negli}(\lambda)$$

which is the desired non-negligible ε'.

But Eq. 6 easily follows from the semantic security of the homomorphic encryption scheme. Indeed assume that we could distinguish between $\mathcal{H}_A^k$ and $\mathcal{H}_A^{k-1}$, then we can decide the following problem, which is easily reducible to the semantic security of $\mathcal{E}$:

Security of $\mathcal{E}$ with respect to Yao Garbled Circuits: *Given a Yao-garbled circuit PK_{Yao}, an input x for it, a random public key $PK_{\mathcal{E}}$ for the homomorphic encryption scheme, a set of ciphertexts $c_1,\ldots,c_n$ where n is the size of x, decide if for all i, $c_i = \mathbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, w_i^{x_i})$, where w_i is the i^{th} input wire and x_i is the i^{th} input bit of x, or c_i is the encryption of a random value.*

Now run experiment $\mathcal{H}_A^{k-1}$ with the following modification: at the k^{th} query, instead of choosing a fresh random key for $\mathcal{E}$ and encrypting random labels, answer with $PK_{\mathcal{E}}$ and the ciphertexts $c_1,\ldots,c_n$ defined by the problem above. If c_i is the encryption of a random value, then we are still running experiment $\mathcal{H}_A^{k-1}$, but if $c_i = \mathbf{Encrypt}_{\mathcal{E}}(PK_{\mathcal{E}}, w_i^{x_i})$, then we are actually running experiment $\mathcal{H}_A^k$. Therefore we can decide the Security of $\mathcal{E}$ with respect to Yao Garbled Circuits with the same advantage with which we can distinguish between $\mathcal{H}_A^k$ and $\mathcal{H}_A^{k-1}$.

The reduction of the Security of $\mathcal{E}$ with respect to Yao Garbled Circuits to the basic semantic security of $\mathcal{E}$ is an easy exercise, and details will appear in the final version. ■

4.5 Proof of Input and Output Privacy

Note that for each oracle query the input and the output are encrypted under the homomorphic encryption scheme $\mathcal{E}$. It is not hard to see that the proof of correctness above, easily implies the proof of input and output privacy. For the one-time case, it obviously follows from the security of Yao's two-party protocol. For the general case, it follows from the semantic security of $\mathcal{E}$, and the proof relies on the same hybrid arguments described above.

4.6 Efficiency

The protocol we have described meets the efficiency goals outlined in Section 3.3. During the preprocessing stage, the client performs $O(|C|)$ work to prepare the Garbled Yao circuit. For each invocation of **ProbGen**, the client generates a new keypair and encrypts one Yao label for each bit of the input, which requires $O(n)$ effort. The worker computes its way through the circuit by performing a constant amount of work per gate, so the worker takes time linear in the time to evaluate the original circuit, namely $O(|C|)$. Finally, to verify the worker's response, the client performs a single decryption and comparison operation for each bit of the output, for a total effort of $O(m)$. Thus, amortized over many inputs, the client performs $O(n+m)$ work to prepare and verify each input and result.

5 How to Handle Cheating Workers

Our definition of security (Definition 3) assumes that the adversary does not see the output of the **Verify** procedure run by the client on the value σ returned by the adversary. Theorem 1 is proven under the same assumption. In practice this means that our protocol $\mathcal{VC}$ is secure if the client keeps the result of the computation private.

In practice, there might be circumstances where this is not feasible, as the behavior of the client will change depending on the result of the evaluation (e.g., the client might refuse to pay the worker). Intuitively, and we prove this formally below, seeing the result of **Verify** on proofs the adversary correctly **Compute**s using the output of **PubProbGen** does not help the adversary (since it already knows the result based on the inputs it supplied to **PubProbGen**). But what if the worker returns a malformed response – i.e., something for which **Verify** outputs $\perp$. How does the client respond, if at all? One option is for the client to ask the worker to perform the computation again. But this repeated request informs the worker that its response was malformed, which is an additional bit of information that a cheating worker might exploit in its effort to generate forgeries. Is our scheme secure in this setting? In this section, we prove that our scheme remains secure as long as the client terminates after detecting a malformed response. We also consider the interesting question of whether our scheme is secure if the client terminates only after detecting $k > 1$ malformed responses, but we are unable to provide a proof of security in this setting.

Note that there is a real attack on the scheme in this setting if the client does not terminate. Specifically, for concreteness, suppose that each ciphertext output by $\textbf{Encrypt}_{\mathcal{E}}$ encrypts a single bit of a label for an input wire of the garbled circuit, and that the adversary wants to determine the first bit $w_{11}^{b_1}$ of the first label (where that label stands in for unknown input $b_1 \in \{0,1\}$). To do this, the adversary runs **Compute** as before, obtaining ciphertexts that encrypt the bits $\overline{w_i}$ of a label for the output wire. Using the homomorphism of the encryption scheme $\mathcal{E}$, it XORs $w_{11}^{b_1}$ with the first bit of $\overline{w_i}$ to obtain $\overline{w_i'}$, and it sends (the encryption of) $\overline{w_i'}$ as its response. If **Verify** outputs $\perp$, then $w_{11}^{b_1}$ must have been a 1; otherwise, it is a 0 with overwhelming probability. The adversary can thereby learn the labels of the garbled circuit one bit at a time – in particular, it can similarly learn the labels of the output wire, and thereafter generate a verifiable response without actually performing the computation.

Intuitively, one might think that if the client terminates after detecting k malformed responses, then the adversary should only be able to obtain about k bits of information about the garbled circuit before the client terminates (using standard entropy arguments), and therefore it should still be hard for the adversary to output the entire "wrong" label for the output wire as long as λ is sufficiently larger than k. However, we are unable to make this argument go through. In particular, the difficulty is with the hybrid argument in the proof of Theorem 1, where we gradually transition to an experiment in which the simulator is encrypting the same Yao input labels in every round. This experiment must be indistinguishable from the real world experiment, which permits different inputs in different rounds. When we don't give the adversary information about whether or not its response was well-formed or not, the hybrid argument is straightforward – it simply depends on the semantic security of the FHE scheme.

However, if we do give the adversary that information, then the adversary can easily distinguish rounds with the same input from rounds with random inputs. To do so, it chooses some "random" predicate P over the input labels, such that $P(w^1_{b_1}, w^2_{b_2}, \ldots) = P(w^1_{b'_1}, w^2_{b'_2}, \ldots)$ with probability $1/2$ if $(b_1, b_2, \ldots) \neq (b'_1, b'_2, \ldots)$. Given the encryptions of $w^1_{b_1}, w^2_{b_2}, \ldots$, the adversary runs **Compute** as in the scheme, obtaining ciphertexts that encrypt the bits $\overline{w_i}$ of a label for the output wire, XORs (using the homomorphism) $P(w^1_{b_1}, w^2_{b_2}, \ldots)$ with the first bit of $\overline{w_i}$, and sends (an encryption of) the result $\overline{w'_i}$ as its response. If the client is making the same query in every round – i.e., the Yao input labels are the same every time – then, the predicate always outputs the same bit, and thus the adversary gets the same response (well-formed or malformed) in every round. Otherwise, the responses will tend to vary.

One could try to make the adversary's distinguishing attack more difficult by (for example) trying to hide which ciphertexts encrypt the bits of which labels – i.e., via some form of obfuscation. However, the adversary may define its predicate in such a way that it "analyzes" this obfuscated circuit, determines whether two ostensibly different inputs in fact represent the same set of Yao input labels, and outputs the same bit if they do. (It performs this analysis on the encrypted inputs, using the homomorphism.) We do not know of any way to prevent this distinguishing attack, and suspect that preventing it may be rather difficult in light of Barak et al.'s result that there is no general obfuscator [7].

Security with Verification Access. We say that a verifiable computation scheme is secure *with verification access* if the adversary is allowed to see the result of **Verify** over the queries x_i he has made to the **ProbGen** oracle in $\mathbf{Exp}^{Verif}_A$ (see Definition 3).

Let $\mathcal{VC}^{\dagger}$ be like $\mathcal{VC}$, except that the client terminates if it receives a malformed response from the worker. Below, we show that $\mathcal{VC}^{\dagger}$ is secure with verification access. In other words, it is secure to provide the worker with verification access (indicating whether a response was well-formed or not), until the worker gives a malformed response. Let $\mathbf{Exp}^{Verif^{\dagger}}_A \left[\mathcal{VC}^{\dagger}, F, \lambda\right]$ denote the experiment described in Section 3.1, with the obvious modifications.

Theorem 4 *If $\mathcal{VC}$ is a secure outsourceable verifiable computation scheme, then $\mathcal{VC}^{\dagger}$ is a secure outsourceable verifiable computation scheme with verification access. If $\mathcal{VC}$ is private, then so is $\mathcal{VC}^{\dagger}$.*

Proof of Theorem 4: Consider two games between a challenger and an adversary A. In the real world game for $\mathcal{VC}^{\dagger}$, Game 0, the interactions between the challenger and A are exactly like those between the client and a worker in the real world – in particular, if A's response was well-formed, the challenger tells A so, but the challenger immediately aborts if A's response is malformed. Game 1 is identical to Game 0, except that when A queries **Verify**, the challenger always answers with the correct y, whether A's response was well-formed or not, and the challenger never aborts. Let ε_i be A's success probability in Game i.

First, we show that if $\mathcal{VC}$ is secure, then ε_1 must be negligible. The intuition is simple: since the challenger always responds with the correct y, there is actually no information in these responses, since A could have computed y on its own. More formally, there is an algorithm B that breaks $\mathcal{VC}$ with probability ε_1 by using A as a sub-routine. B simply forwards communications between the challenger (now a challenger for the $\mathcal{VC}$ game) and A, except that B tells A the correct y w.r.t. all of A's responses. B forwards A's forgery along to the challenger.

Now, we show that $\varepsilon_0 \leq \varepsilon_1$, from which the result follows. Let E_{mal} be the event that A makes a malformed response, and let E_f be the event that A successfully outputs a forgery – i.e., where $\mathbf{Exp}_A^{Verif^\dagger}[\mathcal{VC}^\dagger, F, \lambda]$ outputs '1'. A's success probability, in either Game 0 or Game 1, is:

$$Prob[E_f] = Prob[E_f | E_{mal}] \cdot Prob[E_{mal}] + Prob[E_f | \neg E_{mal}] \cdot Prob[\neg E_{mal}] \tag{7}$$

If A does not make a malformed response, then Games 0 and 1 are indistinguishable to A; therefore, the second term above has the same value in Games 0 and 1. In Game 0, $Prob[E_f | E_{mal}] = 0$, since the challenger aborts. Therefore, $\varepsilon_0 \leq \varepsilon_1$. ■

In practice Theorem 4 implies that every time a malformed response is received, the client must re-garble the circuit (or, as we said above, make sure that the results of the verification procedure remain secret). Therefore the amortized efficiency of the client holds only if we assume that malformed responses do not happen very frequently.

In some settings, it is not necessary to inform the worker that its response is malformed, at least not immediately. For example, in the Folding@Home application [2], suppose the client generates a new garbled circuit each morning for its many workers. At the end of the day, the client stops accepting computations using this garbled circuit, and it (optionally) gives the workers information about the well-formedness of their responses. (Indeed, the client may reveal all of its secrets for that day.) In this setting, our previous security proof clearly holds even if there are arbitrarily many malformed responses.

6 Conclusions and Future Directions

In this work, we introduced the notion of Verifiable Computation as a natural formulation for the increasingly common phenomenon of outsourcing computational tasks to untrusted workers. We describe a scheme that combines Yao's Garbled Circuits with a fully-homomorphic encryption scheme to provide extremely efficient outsourcing, even in the presence of an adaptive adversary. As an additional benefit, our scheme maintains the privacy of the client's inputs and outputs.

Our work leaves open several interesting problems. It would be desirable to devise a verifiable computation scheme that used a more efficient primitive than fully-homomorphic encryption. Similarly, it seems plausible that a verifiable scheme might sacrifice input privacy to increase its efficiency. While our scheme is resilient against a single malformed response from the worker, ideally we would like a scheme that tolerates $k > 1$ malformed responses. Finally, it would be interesting to enhance a verifiable computation scheme to include a non-repudiation property, so that a client who receives a malformed response from a worker can demonstrate the worker's misbehavior to a third party.

References

[1] Amazon Elastic Compute Cloud. Online at `http://aws.amazon.com/ec2`.

[2] The Folding@home project. Stanford University, `http://www.stanford.edu/group/pandegroup/cosm/`.

[3] Sun Utility Computing. Online at `http://www.sun.com/service/sungrid/index.jsp`.

[4] The Great Internet Mersenne Prime Search. `http://www.mersenne.org/prime.htm`.

[5] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@Home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[6] L. Babai. Trading group theory for randomness. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 421–429, New York, NY, USA, 1985. ACM.

[7] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahay, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Proceedings of CRYPTO*, pages 1–18, 2001.

[8] B. Barak, I. Haitner, D. Hofheinz, and Y. Ishai. Bounded key-dependent message security. 2009.

[9] M. Belenkiy, M. Chase, C. C. Erway, J. Jannotti, A. Küpçü, and A. Lysyanskaya. Incentivizing outsourced computation. In *Proceedings of the Workshop on Economics of Networked Systems (NetEcon)*, pages 85–90, New York, NY, USA, 2008. ACM.

[10] D. Chaum and T. Pedersen. Wallet databases with observers. In *Proceedings of CRYPTO*, 1992.

[11] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[12] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, 2009.

[13] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: interactive proofs for muggles. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, 2008.

[14] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[15] P. Golle and I. Mironov. Uncheatable distributed computations. In *Proceedings of the RSA Conference*, 2001.

[16] S. Hohenberger and A. Lysyanskaya. How to securely outsource cryptographic computations. In *Proceedings of TCC*, 2005.

[17] Y. T. Kalai and R. Raz. Probabilistically checkable arguments. In *Proceedings of CRYPTO*, 2009.

[18] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the ACM Symposium on Theory of computing (STOC)*, pages 723–732, New York, NY, USA, 1992. ACM.

[19] J. Kilian. Improved efficient arguments (preliminary version). In *Proceedings of the International Cryptology Conference on Advances in Cryptology*, pages 311–324, London, UK, 1995. Springer-Verlag.

[20] Y. Lindell and B. Pinkas. A proof of Yao's protocol for secure two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[21] S. Micali. CS proofs (extended abstract). In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 1994.

[22] D. Molnar. The SETI@Home problem. *ACM Crossroads*, 7.1, 2000.

[23] F. Monrose, P. Wyckoff, and A. Rubin. Distributed execution with remote audit. In *Proceedings of ISOC Network and Distributed System Security Symposium (NDSS)*, Feb. 1999.

[24] G. Rothblum. *Delegating Computation Reliably: Paradigms and Constructions*. PhD thesis, Massachusetts Institute of Technology, 2009.

[25] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. VanDoorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of the Symposium on Operating Systems Principals*, 2005.

[26] S. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks (Special Issue on Computer Network Security)*, 31:831–960, 1999.

[27] Trusted Computing Group. Trusted platform module main specification. Version 1.2, Revision 103, July 2007.

[28] A. Yao. Protocols for secure computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 1982.

[29] A. Yao. How to generate and exchange secrets. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 1986.

[30] B. S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.