RMC-PVC: A Multi-Client Reusable Verifiable Computation Protocol

Pascal Lafourcade
Université Clermont-Auvergne, CNRS, Mines de Saint-Étienne, LIMOS
Clermont-Ferrand, France

Gael Marcadet
Université Clermont-Auvergne, CNRS, Mines de Saint-Étienne, LIMOS
Clermont-Ferrand, France

Léo Robert
XLIM, Université de Limoges
Limoges, France

ABSTRACT

The verification of computations performed by an untrusted server is a cornerstone for delegated computations, especially in multi-clients setting where inputs are provided by different parties. Assuming a common secret between clients, a *garbled circuit* offers the attractive property to ensure the correctness of a result computed by the untrusted server while keeping the input and the function private. Yet, this verification can be guaranteed only *once*.

Based on the notion of *multi-key homomorphic encryption* (MKHE), we propose RMC-PVC a multi-client verifiable computation protocol, able to verify the correctness of computations performed by an untrusted server for inputs (encoded for a garbled circuit) provided by multiple clients. Thanks to MKHE, the garbled circuit is reusable an arbitrary number of times. In addition, each client can verify the computation by its own. Compared to a single-key FHE scheme, the MKHE usage in RMC-PVC allows to reduce the workload of the server and thus the response delay for the client. It also enforce the privacy of inputs, which are provided by different clients.

ACM Reference Format:
Pascal Lafourcade, Gael Marcadet, and Léo Robert. 2023. RMC-PVC: A Multi-Client Reusable Verifiable Computation Protocol . In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-April 2, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3555776.3577680

1 INTRODUCTION

Consider public hospitals, who depend on a trusted authority such as some government, which want to perform some statistics over COVID-19 statistics (see Fig. 1). In an ideal world, each hospital sends its sensitive data as daily contamination statistics to the server, which then returns the valid result. However, a server can be either compromised by an attacker or malicious in the first place. Therefore, hospitals are facing two problems when outsourcing their data: first, an attacker could learn information about the sensitive data provided by the hospitals and secondly, the attacker could modify the behavior of the server in order to compute an invalid result. Thus, it is crucial that both privacy of the data and integrity of the computations be ensured when outsourcing computations. Each hospital depends on the general authority such as a government and hence is honest. However, each hospital's local data must remain private even after the computation of the statistics.

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
SAC '23, March 27-April 2, 2023, Tallinn, Estonia
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9517-5/23/03…$15.00
https://doi.org/10.1145/3555776.3577680

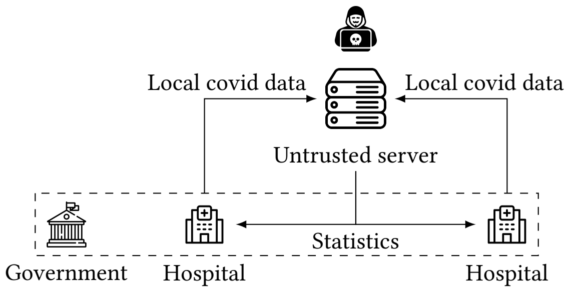


Figure 1: Use-case of RMC-PVC where two hospitals depending on a trusted government want to compute jointly some statistics over local covid data through an untrusted server.

In a more general point of view, ensuring the privacy of data and the correctness of computations when outsourcing computation in a multi-client setting have been intensively studied by many researchers. In [Gennaro et al. 2010], Gennaro *et al.* introduced the notion of *non-interactive verifiable-computation scheme* allowing a client to verify the computation $f(x) = y$ of a function f over an input x, performed by a malicious server. In a verifiable-computation scheme, the server does not perform the function f over the input x but over an encoding of x denoted σ_x. It is possible to retrieve x from σ_x by knowing some secret. Hence, given σ_x only, then server cannot learn information about the input x. The evaluation of the function f over the encoded input σ_x produces the encoding of the result y denoted σ_y. At the end, if the computations done by the server are valid, then the client retrieves $f(x) = y$ from σ_y. Otherwise, the client detects that the computed result is not correct. Note that the server cannot retrieve y from σ_y (as explained in more details in Section 2.1).

The seminal work of Andrew Yao in 1986 introduced the notion of *garbled circuit* [Yao 1986] where non-interactive verifiable-computation scheme was proposed. Garbled circuits ensure the privacy of the input and correctness of computation for a single-evaluation of a boolean circuit f. Some works have shown how to address the one-time limitation such as in [Gennaro et al. 2010] using Fully Homomorphic Encryption (FHE). Introduced in [Gentry 2009], FHE allows a server to perform any computations over encrypted data without relying on the decryption key. The idea presented in [Gennaro et al. 2010] is to evaluate the garbled circuit homomorphically over encrypted encoded input, allowing to reuse the garbled circuit an arbitrary number of times.

The secure function evaluation in multi-client paradigm has led to a variety of approaches. However, compositions of garbled circuits and Multi-Key Homomorphic Encryption (MKHE) have been overlooked. This paper shows that MKHE is interesting for obtaining privacy, and surprisingly, more efficient compared to single-key FHE of [Gennaro et al. 2010].

Contribution. We propose RMC-PVC a Reusable Multi-Client Protocol for Verifiable Computation designed to verify the computations of any function (common to every clients) performed by an untrusted server using garbled circuits, over inputs provided by multiple clients. The usage of a MKHE scheme instead of a single-key FHE is motivated to make the garbled circuit reusable an arbitrary number of times, to ensure input-privacy against both the server and other clients, and to obtain a shorter response delay from the server to the clients by encrypting the garbled circuit *once* (hence not at every computation as done in [Gennaro et al. 2010]). Our protocol uses one key to encrypt the garbled circuit *once* and another key to encrypt the input.

More than the aforementionned properties ensured by our construction, we present a feasability result in combining MKHE and garbled circuits in the multi-client setting. Combining FHE and garbled circuits has already been shown to be feasible in [Gennaro et al. 2010], which relies on a single-key FHE scheme. However, generalizing this approach in the multi-client setting using the same homomorphic encryption key for each client, does *not* prevent a client to learn input of the other clients. In contrast, the usage of a MKHE scheme instead of a single-key FHE scheme, and providing for each client its own encryption key can overcome this problem, but remains an interesting question. However, with MKHE, we expect the result to be encrypted under multiple public encryption keys. In a setting where related secret keys are owned by different clients, the decryption is a *non-trivial question.* To overcome this issue, we use a particular MKHE scheme having a *distributed decryption* property [Chen et al. 2019], allowing each client to verify the untrusted server's computations by its own.

Related Work. The notion of non-interactive verifiable computing, introduced in [Gennaro et al. 2010], have been widely studied and have led to a variety of solutions. The notion *non-interactive* proofs are crucial for looking at correctness of computations over a minimal number of rounds, to reduce as much as possible the communications between clients and the server. Micali's Computational Sound proofs [Micali 2000] are adapted for non-interactive verifiable computations under the Random Oracle Model, as well as [Goldwasser et al. 2015] but only for restricted class of functions. We move away from this paradigm to focus on garbled circuits, allowing to ensure the correctness of computations within two rounds, independently of the size of the function to be evaluated.

The one-time limitation of garbled circuits has been addressed in different approaches, either by using *Cut-And-Choose* [Gennaro et al. 2010; Huang et al. 2014; Lindell and Pinkas 2015] technique, or by using homomorphic encryption [Gennaro et al. 2010], but only in a single-client setting. Multi-client verifiable computation setting has been addressed in numerous ways. For instance, Goldwasser *et al.* used multi-party functional encryption, however restricted to binary function [Goldwasser et al. 2014] whereas our work handles an arbitrary function.

The work of [Kamara et al. 2011] proposed a multi-client verifiable computation scheme where every client shares the same pool of entropy in order to generate the same garbled circuit. The main drawback of this solution is that a new garbled circuit has to be generated for each round of computations, even if the function remains the same. Our work adds a reusability property meaning that the garbled circuit is computed only once for a given function (even if the evaluation is done multiple times). Closer to our work, [Choi et al. 2013] studied a multi-client verifiable computation scheme based on garbled circuit using Proxy Oblivious Transfer, where the first client is in charge of the creation of the garbled circuit and the encoding of the input for the other clients. Rather to create an unbalanced execution time among clients, only the first client verifies computations performed by the server. In our protocol, every client has the same workload and can verify computations by its own.

Functional Encryption (FE) has been shown to be an interesting cryptographic primitive. Indeed, [Goldwasser et al. 2013] has exposed a reusable garbled circuit. In summary, they rely on an universal circuit which given a circuit C and an input x, outputs $C(x)$. The circuit is given to the server, as well as x encrypted using the functional encryption scheme. The universal circuit, viewed as a function, is linked with the decryption key of the FE scheme. To obtain the result, they rely on the decryption function of the FE scheme, by decrypting the encryption of x, producing the desired output. Their approach does not ensure the privacy of the output, in constrast to our construction where the output privacy is ensured.

More recent work such that the construction of [Fiore et al. 2020] proposed to compute the function f homomorphically and later prove the validity of the computation using a Succint Non-interactive Argument of Knowledge. Similarly in [Bois et al. 2021], they suggested a verifiable computation scheme using Somewhat Homomorphic Encryption (a FHE scheme supporting a bounded-degree algrebraic circuit) and an Homomorphic Hash function allowing to operate over hashs. Althrough efficient for the proving and verification, both of these works are limited to the single-client setting. Moreover, work of [Bois et al. 2021] supports only a bounded-degree functions. It contrast our work, designed for the multi-client setting and supporting arbitratry functions.

Close to our work, [Lafourcade et al. 2022] presents a non-interactive verifiable computation based on MKHE and garbled circuits for the single-client setting. Conceptually, the client sends to the server his input σ_x encrypted under an MKHE key pk_x (with sk_x kept secret by the client). Then, given a garbled circuit encrypted under pk_f (with sk_f owned by the client), the server sends back σ_y encrypted under both pk_x and pk_f. Their approach *cannot* be generalized to the multi-client setting: assuming a client sending its input encrypted under its own key, then the obtained encoded result would be encrypted under the public key of all clients. So, the decryption is impossible without sharing the secret keys to other clients, thus exposing the inputs. Our paper proposes a solution for the multi-client setting where σ_y can be obtained by other clients without exposing their secret keys thus ensuring the input privacy.

Outline. We recall in Section 2 the definition of a verifiable computation scheme as well as the building blocks used for RMC-PVC. In Section 3, we detail the design of RMC-PVC. Due to the lack of space, we present in Section 4 an overview of the security and refer the reader on the long version [RMC 2022] for the proofs. Finally, we analyse the complexity of our scheme in Section 5.

2 PRELIMINARIES

We present the definition of a verifiable-computation scheme in the general setting. We also describe the core primitives of RMC-PVC, namely the Yao's garbled circuit and MKHE scheme.

2.1 Verifiable Computation Scheme

A verifiable computation scheme aims to verify outsourced computations over an untrusted server. Suppose that a client wants to delegate $y = f(x)$ the computation of the function f over an input x to a server. We provide the model of the verifiable computation defined in [Gennaro et al. 2010]:

Definition 2.1 (Verifiable Computation Scheme [Gennaro et al. 2010]). A *verifiable computation scheme* VC is composed of four algorithms VC.KeyGen, VC.ProbGen, VC.Compute, VC.Verify:

- $\mathsf{VC.KeyGen}(f, \lambda) \rightarrow (\mathsf{PK}, \mathsf{SK})$: Given a function f and a security parameter λ, output public key PK (that encodes the function f) and secret key SK.
- $\mathsf{VC.ProbGen}_{\mathsf{SK}}(x) \rightarrow (\sigma_x, \tau_x)$: Given the input x and the private key SK, return σ_x, the public encoding of input x, (used by the server to compute $y = f(x)$) and τ_x, a secret verification string (used to verify correctness of computations).
- $\mathsf{VC.Compute}_{\mathsf{PK}}(\sigma_x) \rightarrow \sigma_y$: Given the public key PK and the encoded input σ_x, return σ_y the encoded output.
- $\mathsf{VC.Verify}_{\mathsf{SK}}(\sigma_y, \tau_x) \rightarrow y \cup \perp$: Given SK the private key, τ_x the secret verification string and σ_y the encoded output, output y if the output is verified, $\perp$ otherwise.

A verifiable computation scheme VC must be *correct*, *secure* and *outsourceable.* A VC scheme is *correct* for any input x, an honest server can compute $f(x)$. A VC scheme is *secure* if the client is able to detect if a malicious server returns $\hat{y}$ an invalid result instead of the valid result $f(x)$. A VC scheme is *outsourceable* meaning that the client must perform strictly less operations than computing the function by himself. We give the formal definitions of usual security properties of a verifiable computation scheme. Optionally, a VC scheme is said *private* if the server cannot learn some information about the inputs x and the output $f(x)$.

Definition 2.2 (Correctness [Gennaro et al. 2010]). Let VC be a verifiable computation scheme. For every function f, every input x, given $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{VC.KeyGen}(f, \lambda)$, the equality holds:

$$f(x) = \mathsf{VC.Verify}_{\mathsf{SK}}(\mathsf{VC.Compute}_{\mathsf{PK}}(\mathsf{VC.ProbGen}_{\mathsf{SK}}(x)))$$

Definition 2.3 (Outsourceability). For every function f, a verifiable computation scheme VC is *outsourceable* if the asymptotic complexities of VC.ProbGen and VC.Verify are strictly lower than the asymptotic complexity of the fastest algorithm to compute f.

A verifiable computation scheme is *secure* if the server manages to return an invalid result accepted by the client only with a negligible probability. We formalize the security of the verifiable computation scheme with the game $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{Sec}}$ presented in Fig. 2; the challenger expects from the adversary to produce invalid encoded result $\sigma_{\hat{y}}$ where $\hat{y} \neq f(x)$ for an input x chosen by the adversary. The adversary must also produce σ_x the encoding of the input x. The adversary has access to an oracle $O^{\mathsf{PubProbGen}}$ which for an input x, computes the encoding σ_x as well as the private encoding τ_x before to output only σ_x. The oracle $O^{\mathsf{PubProbGen}}$ can be called by the adversary a polynomial number of times and until to output the result. The adversary wins at $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{Sec}}$ if the provided encoded result is accepted by the verifiable computation scheme.

```
Game $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{Sec}}[\mathsf{VC}, f, \lambda]$:
    $\mathcal{T} \leftarrow \emptyset$
    $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{VC.KeyGen}(f, \lambda)$
    $x, \sigma_x, \sigma_{\hat{y}} \leftarrow \mathcal{A}^{O^{\mathsf{PubProbGen}}(\mathcal{T}, \mathsf{VC}, \mathsf{SK}, \cdot)}(\mathsf{PK})$
    retrieve $\tau_x$ associated with $\sigma_x$ from $\mathcal{T}$
    $\hat{y} \leftarrow \mathsf{VC.Verify}_{\mathsf{SK}}(\sigma_{\hat{y}}, \tau_x)$
    Outputs 1 if $\hat{y} \neq \perp \bigwedge \hat{y} \neq f(x)$ else 0

Oracle $O^{\mathsf{PubProbGen}}(\underline{\mathcal{T}, \mathsf{VC}, \mathsf{SK}}, x)$:

    $(\sigma_x, \tau_x) \leftarrow \mathsf{VC.ProbGen}_{\mathsf{SK}}(x)$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{(\sigma_x, \tau_x)\}$
    Outputs $\sigma_x$
```

Figure 2: Experiment for *security* of a verifiable computation scheme. By $\mathsf{poly}(\lambda)$, we denote a polynomial function depending on the security parameter λ. Parameters provided by the challenger are underlined.

Definition 2.4 (Security [Gennaro et al. 2010]). Let VC be a verifiable computation scheme and λ the security parameter, then for every function f, and every $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{VC.KeyGen}(f, \lambda)$, for every PPT adversary $\mathcal{A}$ we have: $\Pr[\mathsf{Exp}^{\mathcal{A}}_{\mathsf{Sec}}[\mathsf{VC}, f, \lambda] = 1] = \mathsf{negl}(\lambda)$.

A verifiable computation scheme is *private* if the server does not learn information about the input, or with a negligible probability. We formalize the privacy of a verifiable computation scheme with the game $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{Priv}}$ introduced in Fig. 3. The challenger chooses one input x_b among x_0 and x_1 provided by the adversary $\mathcal{A}$, where b is a random bit. Then, the challenger computes and sends σ_{x_b} the encoding of the input x_b to $\mathcal{A}$, which responds with $\hat{b}$ a prediction over the bit b. The adversary $\mathcal{A}$ wins at $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{Priv}}$ if his prediction $\hat{b}$ is valid *i.e.,* $\hat{b} = b$. Before the prediction, $\mathcal{A}$ can call the oracle $O^{\mathsf{PubProbGen}}$ (defined in Fig. 2) a polynomial number of times.

Definition 2.5 (Privacy [Gennaro et al. 2010]). Let VC be a verifiable computation scheme and λ a security parameter. Then, for every function f and every $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{VC.KeyGen}(f, \lambda)$, for every PPT adversary $\mathcal{A}$ we have $\Pr[\mathsf{Exp}^{\mathcal{A}}_{\mathsf{Priv}}[\mathsf{VC}, f, \lambda] = 1] = \frac{1}{2} + \mathsf{negl}(\lambda)$.

2.2 Yao's Garbled Circuit

Suppose that a client wants to delegate $f(x)$, the computation of a function f over an input x to an untrusted server. The first step performed by the client is to compute $\Delta_f = (\mathsf{G}, \mathcal{W}, \mathcal{W}_{\mathrm{in}}, \mathcal{W}_{\mathrm{out}})$ the boolean circuit corresponding to f, where G is the set of gates in Δ_f, $\mathcal{W}$ is the set of wires in Δ_f, $\mathcal{W}_{\mathrm{in}}$ (resp. $\mathcal{W}_{\mathrm{out}}$) are the wires at the input (resp. output) of Δ_f, with $\mathcal{W}_{\mathrm{in}} \subset \mathcal{W}$ (resp. $\mathcal{W}_{\mathrm{out}} \subset \mathcal{W}$). Each gate $g \in \mathsf{G}$ (see Fig. 4) takes as input two wires $w_0, w_1 \in \mathcal{W}$ and as an output a wire $w_2 \in \mathcal{W}$, denoted by $g(w_0, w_1) \rightarrow w_2$.

```
Game $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{Priv}}[\mathsf{VC}, f, \lambda]$

  $\mathcal{T} \leftarrow \emptyset$
  $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{VC.KeyGen}(f, \lambda)$
  $(x_0, x_1) \leftarrow \mathcal{A}^{O^{\mathsf{PubProbGen}}(\mathcal{T}, \mathsf{VC}, \mathsf{SK}, \cdot)}(\mathsf{PK})$
  $b \xleftarrow{\$} \{0, 1\}$
  $(\sigma_{x_b}, \tau_{x_b}) \leftarrow \mathsf{VC.ProbGen}_{\mathsf{SK}}(x_b)$
  $\hat{b} \leftarrow \mathcal{A}(\sigma_{x_b})$

  Outputs 1 if $b = \hat{b}$ else 0
```

Figure 3: Experiment for *privacy* property of a verifiable computation scheme. Parameters provided by the challenger are underlined.

To generate the garbled circuit, we require a symmetric encryption scheme denoted Sym composed of three functions Sym.KeyGen, Sym.Enc and Sym.Dec:

- $\mathsf{Sym.KeyGen}(1^\lambda) \rightarrow k$: Given a security parameter λ, output a λ-string k.
- $\mathsf{Sym.Enc}_k(m) \rightarrow \mathsf{c}$: Given a message m, output ct the message m encrypted under the key k.
- $\mathsf{Sym.Dec}_k(\mathsf{c}) \rightarrow m$: Given a ciphertext c representing the message m encrypted under the key k, output m.

Garbled circuit relies on a *Yao-Secure* symmetric encryption scheme.

Definition 2.6 (Yao-Secure Symmetric Encryption [Yao 1986]). A *Yao-Secure* symmetric encryption scheme must ensure:

- *Indistinguishable under multiple messages*, meaning that for every two vectors of polynomial length $\bar{x}$ and $\bar{y}$, no polynomial time adversary can distinguish between an encryption of $\bar{x}$ or $\bar{y}$ under key k.
- *Elusive range* meaning that the encryption of a message falls under different key spaces depending on the encryption key used.
- *Efficient checkable range* where given a key k and a ciphertext c, there is a polynomial time algorithm able to check if c has been encrypted under k.

For each wire $w \in \mathcal{W}$, the client chooses uniformly random *labels* $k_w^0 \leftarrow \mathsf{Sym.KeyGen}(1^\lambda)$, $k_w^1 \leftarrow \mathsf{Sym.KeyGen}(1^\lambda)$ corresponding respectively to 0 (wire off) and 1 (wire on). Once each wire has been affected with two labels, then for each gate $g(w_0, w_1) \rightarrow w_2$ we compute γ_g as follows:

$$\begin{aligned}
\gamma_g &= (\gamma_g^{00}, \gamma_g^{01}, \gamma_g^{10}, \gamma_g^{11}) \\
\gamma_g^{00} &= \mathsf{Sym.Enc}_{k^0_{w_0}}(\mathsf{Sym.Enc}_{k^0_{w_1}}(k_{w_2}^{g(0,0)})) \\
\gamma_g^{01} &= \mathsf{Sym.Enc}_{k^0_{w_0}}(\mathsf{Sym.Enc}_{k^1_{w_1}}(k_{w_2}^{g(0,1)})) \\
\gamma_g^{10} &= \mathsf{Sym.Enc}_{k^1_{w_0}}(\mathsf{Sym.Enc}_{k^0_{w_1}}(k_{w_2}^{g(1,0)})) \\
\gamma_g^{11} &= \mathsf{Sym.Enc}_{k^1_{w_0}}(\mathsf{Sym.Enc}_{k^1_{w_1}}(k_{w_2}^{g(1,1)}))
\end{aligned} \tag{1}$$

From Equation (1), we construct the *garbled circuit* $\gamma = \{\gamma_g | g \in \mathsf{G}\}$.

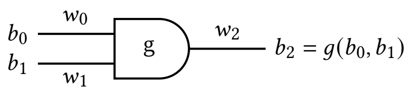


Figure 4: Representation of the gate $g(w_0, w_1) \rightarrow w_2 \in \mathsf{G}$.

Definition 2.7 (Yao's garbled circuit[Yao 1986]). Suppose Sym a Yao-Secure symmetric encryption scheme. The verifiable computation scheme VCYao is composed of VCYao.KeyGen, VCYao.ProbGen, VCYao.Compute, VCYao.Verify where:

- $\mathsf{VCYao.KeyGen}(f, \lambda) \rightarrow (\mathsf{PK}, \mathsf{SK})$: Compute the boolean circuit $\Delta_f = (\mathsf{G}, \mathcal{W}, \mathcal{W}_{\text{in}}, \mathcal{W}_{\text{out}})$ corresponding to f, where G is the set of gates, $\mathcal{W}$ the set of wires, $\mathcal{W}_{\text{in}}$ the set of input wires and $\mathcal{W}_{\text{out}}$ the set of output wires. For each wire $w \in \mathcal{W}$, compute $k_w^0 \leftarrow \mathsf{Sym.KeyGen}(1^\lambda)$, $k_w^1 \leftarrow \mathsf{Sym.KeyGen}(1^\lambda)$. Compute the *garbled circuit* $\gamma = \{\gamma_g | g \in \mathsf{G}\}$. Output $\mathsf{PK} = \gamma$ and $\mathsf{SK} = \cup_{w \in \mathcal{W}} \{k_w^0, k_w^1\}$.
- $\mathsf{VCYao.ProbGen}_{\mathsf{SK}}(x) \rightarrow \sigma_x$: We denote the binary representation of input x composed of n bits by $x_1, \ldots, x_n$. Output $\sigma_x = \{k_{w_1}^{x_1}, \ldots, k_{w_n}^{x_n}\}$ the set of labels associated to the wire $w_i \in \mathcal{W}_{\text{in}}$ regarding on the input bits.
- $\mathsf{VCYao.Compute}_{\mathsf{PK}}(\sigma_x) \rightarrow \sigma_y$: Output $\sigma_y = \{k_{w_1}^{y_1}, \ldots, k_{w_m}^{y_m}\}$ the set of labels representing the binary encoding $y_1, \ldots, y_m$ of the output $y = f(x)$ using the garble circuit γ (simulating the function f) over the encoded input σ_x.
- $\mathsf{VCYao.Verify}_{\mathsf{SK}}(\sigma_y) \rightarrow y \cup \perp$: Output $y = y_1, \ldots, y_m$ only if for all $i \in [1, m]$ we have $k_{w_i}^{y_i} \in \{k_{w_i}^0, k_{w_i}^1\}$, otherwise the server is cheating, thus we refuse the result with $\perp$.

2.3 Multi-Key Homomorphic Encryption

Introduced in 2009 by Gentry [Gentry 2009], the Fully Homomorphic Encryption (FHE) allows to perform any circuit over encrypted data. A traditional use-case of FHE starts by a client who sends to a server an input x encrypted under a public key *pk*. Then, the server sends back the result of the function f over x, without relying on the decryption key *sk*. Multi-Key Homomorphic Encryption (MKHE) is a natural extension of FHE where the evaluation of a circuit is performed over inputs encrypted under different keys. In this work, we consider the MKHE scheme in [Chen et al. 2017].

Definition 2.8 (Multi-Key Homomorphic Encryption). A MKHE scheme with MKHE.KeyGen, MKHE.Enc, MKHE.Eval, MKHE.Dec:

- $\mathsf{MKHE.KeyGen}(1^\lambda) \rightarrow (pk, sk)$: Given the unary representation of the security parameter λ, output a new key pair (pk, sk) with *pk* the public key and *sk* the secret key.
- $\mathsf{MKHE.Enc}(pk, m) \rightarrow \{m\}_{\mathsf{PK}}$: Given message m and public key *pk*, output $\{m\}_{\mathsf{PK}}$ the message m encrypted under the set of public keys PK containing exactly one public key *pk*.
- $\mathsf{MKHE.Eval}(f, \{m_0\}_{\mathsf{PK}_0}, \{m_1\}_{\mathsf{PK}_1}) \rightarrow \{f(m_0, m_1)\}_{\mathsf{PK}_0 \cup \mathsf{PK}_1}$: Given a boolean circuit f, a ciphertext $\{m_0\}_{\mathsf{PK}_0}$ containing the message m_0 encrypted under the set of public keys PK_0, a ciphertext $\{m_1\}_{\mathsf{PK}_1}$ containing message m_1 encrypted under the set of public keys PK_1, output $\{f(m_0, m_1)\}_{\mathsf{PK}_0 \cup \mathsf{PK}_1}$ the evaluation of function f over the inputs m_0 and m_1 encrypted under the union of the sets of public keys PK_0 and PK_1.

- $\mathsf{MKHE.Dec}(\mathsf{SK}, \{m\}_{\mathsf{PK}}) \rightarrow m$: Given the set of secret keys SK and a ciphertext $\{m\}_{\mathsf{PK}}$ containing a message m encrypted under the set of public keys PK, output m.

In this paper, we consider the MKHE scheme of [Chen et al. 2019] having the *distributed decryption* property. Conceptually, given a ciphertext and its secret key, a party can extract a so-called partial decryption. To obtain the plaintext, each party requires the partial decryption of the other parties. Then, by merging all partial decryptions, the party recovers the underlying plaintext.

Definition 2.9 (Distributed Decryption[Chen et al. 2019]). Suppose the set of public encryption keys $\mathsf{PK} = \{pk_0, \dots, pk_n\}$. Distributed decryption property of a MKHE scheme considers two additional functions MKHE.PartDec and MKHE.Merge where:

- $\mathsf{MKHE.PartDec}(sk_i, \{m\}_{\mathsf{PK}}) \rightarrow \mathsf{Part}_i(m)$: Given a secret key sk_i and a ciphertext $\{m\}_{\mathsf{PK}}$ containing a message m encrypted under the set of public keys PK, output $\mathsf{Part}_i(m)$ a partial decryption of $\{m\}_{\mathsf{PK}}$ using sk_i.
- $\mathsf{MKHE.Merge}(\mathsf{Part}_0(m), \dots, \mathsf{Part}_n(m)) \rightarrow m$: Given partial decryption $\mathsf{Part}_i(m)$ where $i \in \{0, \dots, n\}$, output m.

For clarity, given a ciphertext $c = \{m\}_{\mathsf{PK}}$ a message m encrypted under a *short* set of public keys *e.g.,* $\mathsf{PK} = \{pk_0, pk_1\}$, we denote c directly by $\{m\}_{pk_0, pk_1}$.

```
Game $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{MKICDD}}[\lambda]$ :

  $\mathcal{K} \leftarrow \emptyset$
  $(pk_m, sk_m) \leftarrow \mathsf{MKHE.KeyGen}(1^\lambda)$
  $x_0, x_1 \leftarrow \mathcal{A}^{O^{\mathsf{Enc}}(\mathcal{K},\lambda,\cdot), O^{\mathsf{PartDec}}(\mathcal{K},pk_m,\cdot,\cdot)}(pk_m)$
  $b \leftarrow\$ \{0, 1\}$
  $\{x_b\}_{pk_m} \leftarrow \mathsf{MKHE.Enc}(pk_m, x_b)$
  $b' \leftarrow \mathcal{A}^{O^{\mathsf{Enc}}(\mathcal{K},\lambda,\cdot), O^{\mathsf{PartDec}}(\mathcal{K},pk_m,\cdot,\cdot)}(pk_m, \{x_b\}_{pk_m})$
  Output 1 if $b = \hat{b}$ else 0

Oracle $O^{\mathsf{Enc}}(\underline{\mathcal{K}, \lambda}, x)$ :

  $(pk, sk) \leftarrow \mathsf{MKHE.KeyGen}(1^\lambda)$
  $\{x\}_{pk} \leftarrow \mathsf{MKHE.Enc}(pk, x)$
  $\mathcal{K} \leftarrow \mathcal{K} \cup \{(pk, sk)\}$

  Output $\{pk, \{x\}_{pk}\}$

Oracle $O^{\mathsf{PartDec}}(\underline{\mathcal{K}, pk_m}, \psi, pk)$ :

  if $pk \neq pk_m$
    Retreive $(pk, sk)$ from $\mathcal{K}$
    Return $\mathsf{MKHE.PartDec}(sk, \psi)$
  else
    Return $\perp$
```

Figure 5: Experiment for the semantic security of a MKHE scheme having the distributed decryption property.

In Fig. 5, we present the game modeling the semantic security of the MKHE scheme having the distributed decryption property. In this game, the adversary $\mathcal{A}$ is called by the challenger to produce two inputs x_0 and x_1 of his choice. The challenger samples a random bit b and encrypts x_b under the key pk_m, denoted $\{x_b\}_{pk_m}$. Then, the adversary $\mathcal{A}$ is called with the public encryption key pk_m and the ciphertext $\{x_b\}_{pk_m}$, which responds with a prediction bit $\hat{b}$. The challenger outputs 1 if $\hat{b}$ equals to b, 0 otherwise. During the game, the adversary $\mathcal{A}$ can call a polynomial number of times two oracles O^{Enc} and O^{PartDec}. Both have in common a set $\mathcal{K}$ allowing to share generated keys among the oracles. Every parameters given by the challenger to the oracles, underlined in the game, are assumed to be hidden from $\mathcal{A}$. The first oracle O^{Enc} takes as an input three parameters. Two first parameters, provided are the set $\mathcal{K}$ and the security parameter λ. The last parameter is a bitstring x provided by the adversary, which expects as an output the encryption of x. The second oracle O^{PartDec} takes as an input four parameters. The two first parameters, provided by the challenger, are the set $\mathcal{K}$ and the public encryption key pk_m. The two last parameters, provided by the adversary, are a bistring ψ and a public encryption key pk. The adversary expects from O^{PartDec} to produce a partial decryption of the bistring ψ with sk (obtained by the oracle by searching in $\mathcal{K}$ with pk). If the adversary sends either a malformed ciphertext ψ or public encryption key pk or a pk not contained in $\mathcal{K}$, then the oracle returns $\perp$ and the adversary would not get any advantage.

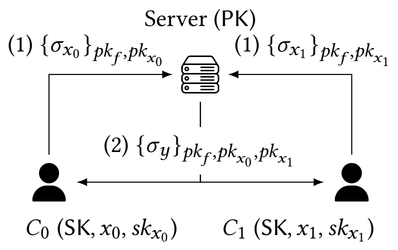
3 OUR PROTOCOL RMC-PVC

We present our verifiable-computation protocol called RMC-PVC (for Reusable Multi-Clients Protocol for Verifiable Computing) designed to verify computations performed by an untrusted server of an arbitrary function f over inputs provided by multiple clients. RMC-PVC is non-interactive: Communications between the server and the clients occurs during the input providing, and during the result verification steps.

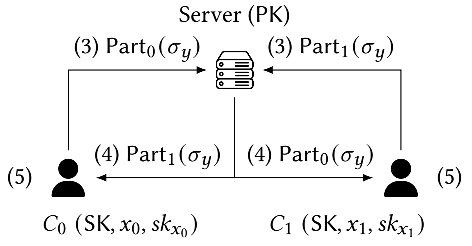
For the sake of clarity, we present our protocol RMC-PVC in a two-clients setting. We stress that RMC-PVC works for an *arbitrary* number of clients. After the presentation for two clients, we show how to generalize our protocol for any number of clients. Still in the search of clarity, we assume that one client provides one bit of information, without loss of generalization.

3.1 Initial Setup

The initial setup of our protocol requires the knowledge of the key pair $(\mathsf{PK}, \mathsf{SK})$ by clients C_0 and C_1. In our use-case, the key pair $(\mathsf{SK}, \mathsf{PK})$ is initially computed by a trusted honest party such as the government, and then shared to hospitals. The key pair is constructed as follows: Assume a boolean circuit f represented by the tuple $(\mathsf{G}, \mathcal{W}, \mathcal{W}_{\text{in}}, \mathcal{W}_{\text{out}})$ where G is the set of gates, $\mathcal{W}$ the set of wires, $\mathcal{W}_{\text{in}} \subset \mathcal{W}$ the set of input wires and $\mathcal{W}_{\text{out}} \subset \mathcal{W}_{\text{in}}$ the set of output wires. The trusted authority computes $(\mathsf{PK}_{\mathsf{Yao}}, \mathsf{SK}_{\mathsf{Yao}}) \leftarrow \mathsf{VCYao.KeyGen}(f, \lambda)$ where $\mathsf{PK}_{\mathsf{Yao}} = \{\gamma\} = \{\gamma_g | g \in \mathsf{G}\}$ and $\mathsf{SK}_{\mathsf{Yao}} = \cup_{w \in \mathcal{W}} \{k^0_w, k^1_w\}$. Then, it generates the function-related key pair $(pk_f, sk_f) \leftarrow \mathsf{MKHE.KeyGen}(1^\lambda)$, and finally computes the encrypted garbled circuit denoted $\{\gamma\}_{pk_f}$, which is equivalent to $\{\mathsf{MKHE.Enc}(pk_f, \gamma_g) | g \in \mathsf{G}\}$. By $\{0\}_{pk_f}$, we denote the bit 0 encrypted with pk_f. We set $\mathsf{PK} = \{\{\gamma\}_{pk_f}\}$ and $\mathsf{SK} = \mathsf{SK}_{\mathsf{Yao}} \cup \{sk_f\}$. In the general case with multi-clients, all clients must share SK.


Server (PK)
(1) $\{\sigma_{x_0}\}_{pk_f, pk_{x_0}}$
(1) $\{\sigma_{x_1}\}_{pk_f, pk_{x_1}}$
(2) $\{\sigma_y\}_{pk_f, pk_{x_0}, pk_{x_1}}$
C_0 (SK, x_0, sk_{x_0})
C_1 (SK, x_1, sk_{x_1})

(a) Computation of $y = f(x_0, x_1)$ where the function f is simulated by the server using PK over σ_{x_0} and σ_{x_1} the encoding of respectively the input x_0 and x_1.


Server (PK)
(3) $\text{Part}_0(\sigma_y)$
(3) $\text{Part}_1(\sigma_y)$
(4) $\text{Part}_1(\sigma_y)$
(4) $\text{Part}_0(\sigma_y)$
(5)
(5)
C_0 (SK, x_0, sk_{x_0})
C_1 (SK, x_1, sk_{x_1})

(b) Decryption of the encrypted encoded result $\{\sigma_y\}_{pk_f, pk_{x_0}, pk_{x_1}}$, where each client shares his own partial decryption to other clients through the server.

Figure 6: Overview of RMC-PVC. Computation is presented in Figure 6a. Decryption and verification is presented in Figure 6b.

3.2 RMC-PVC with Two Clients

Suppose two clients C_0 and C_1 having respectively inputs x_0 and x_1, and want the server to compute $y = f(x_0, x_1)$ with $f : \{0,1\}^2 \to \{0,1\}^m$. RMC-PVC works in five steps. Steps (1) and (2) are presented in Fig. 6a while Steps (3), (4) and (5) are depicted in Fig. 6b.

Step (1): Inputs encoding. Given his input x_0, client C_0 computes $(\sigma_{x_0}, \tau_{x_0})$ by executing $\text{VCYao.ProbGen}_{\text{SK}_{\text{Yao}}}(x_0)$. Then, client C_0 generates the *input key pair* (pk_{x_0}, sk_{x_0}) with $\text{MKHE.KeyGen}(1^\lambda)$, dedicated for input x_0. The public encoding σ_{x_0} of the input x_0 is encrypted under both pk_f and pk_{x_0}, producing $\{\sigma_{x_0}\}_{pk_f, pk_{x_0}}$, obtained by doing an homomorphic addition of the ciphertexts $\text{MKHE.Enc}(pk_{x_0}, \sigma_{x_0})$ and $\text{MKHE.Enc}(pk_f, 0)$. Client C_1 does the same operations over its own input x_1 in order to obtain $\{\sigma_{x_1}\}_{pk_f, pk_{x_1}}$. Then, client C_0 (resp. C_1) sends $\{\sigma_{x_0}\}_{pk_f, pk_{x_0}}$ (resp. $\{\sigma_{x_1}\}_{pk_f, pk_{x_1}}$) to the server as depicted in Fig. 6a.

Step (2): Function evaluation. The server evaluates the function f using the encrypted garbled circuit $\{\gamma\}_{pk_f}$ and encrypted encoded inputs $\{\sigma_{x_0}\}_{pk_f, pk_{x_0}}$ and $\{\sigma_{x_1}\}_{pk_f, pk_{x_1}}$, provided by clients at Step (1). The function evaluation assumes a circuit denoted Γ working at level of a gate $g(w_0, w_1) \to w_2 \in \text{G}$, which given two labels $k_{w_0}^{b_0}, k_{w_1}^{b_1}$ (corresponding to bits b_0 and b_1) and the set of ciphertexts γ_g (detailed in Equation (1)), produces $k_{w_2}^{g(b_0,b_1)}$. Executed homomorphically, Γ allows to run a garbled circuit homomorphically without revealing the produced label $k_{w_2}^{g(b_0,b_1)}$, ensuring the reusability of the garbled circuit. Therefore, the server performs the evaluation of the circuit f by executing gate-by-gate $g(w_0, w_1) \to w_2 \in \text{G}$ the homomorphic evaluation function $\text{MKHE.Eval}(\Gamma, \{k_{w_0}^{b_0}\}_{pk_f, pk_{x_0}, pk_{x_1}}, \{k_{w_1}^{b_1}\}_{pk_f, pk_{x_0}, pk_{x_1}}, \{\gamma_g\}_{pk_f})$, which produces the ciphertext $\{k_{w_2}^{g(b_0,b_1)}\}_{pk_f, pk_{x_0}, pk_{x_1}}$ the label $k_{w_2}^{g(b_0,b_1)}$ corresponding to bit $g(b_0, b_1)$ for the wire $w_2 \in \mathcal{W}$, encrypted under pk_f, pk_{x_0} and pk_{x_1}. At the end of the circuit, the server obtains a set of outputs labels $\{\{k_{w_i}^{y_i}\}_{pk_f, pk_{x_0}, pk_{x_1}}\}_m^{i=1}$ denoted as $\{\sigma_y\}_{pk_f, pk_{x_0}, pk_{x_1}}$, sent by as a response to every client (see Fig. 6a).

Step (3): Partial decryption. From $\{\sigma_y\}_{pk_f, pk_{x_0}, pk_{x_1}}$, the client C_0 computes $\text{Part}_0(\sigma_y) \leftarrow \text{MKHE.PartDec}(sk_{x_0}, \{\sigma_y\}_{pk_f, pk_{x_0}, pk_{x_1}})$ its own partial decryption. Client C_1 does the same to obtain the partial decryption $\text{Part}_1(\sigma_y)$, sent to the server, depicted in Fig. 6b.

Step (4): Partial decryptions sharing. In this step, the server is in charge to perform the sharing of partial decryptions. The server sends $\text{Part}_0(\sigma_y)$ the partial decryption (computed and sent by C_0) to C_1. Symmetrically, the server sends $\text{Part}_1(\sigma_y)$ the partial decryption computed by C_1, to C_0 as depicted in Fig. 6b. Note that to retreive σ_y, the server has to merge every partial decryptions, including $\text{Part}_f(\sigma_y)$. However, $\text{Part}_f(\sigma_y)$ cannot be computed by the server since it does not know sk_f kept secret by the clients. Hence, the usage of pk_f is legitimated both for efficiency (*i.e.,* garbled circuit is encrypted once) but also to prevent the server to recover information.

Step (5): Result verification. Before ensuring the integrity of the computation, client C_0 retrieves the encoding of the result returned by the server. To retreive the encoded output σ_y, C_0 requires the last partial decryption related the with the public key pk_f, computed as $\text{Part}_f(\sigma_y) \leftarrow \text{MKHE.PartDec}(sk_f, \{\sigma_y\}_{pk_f, pk_{x_0}, pk_{x_1}})$. Then, C_0 computes $\sigma_y \leftarrow \text{MKHE.Merge}(\text{Part}_f(\sigma_y), \text{Part}_0(\sigma_y), \text{Part}_1(\sigma_y))$. Then, to verify the encoding σ_y, C_0 computes the output $y \leftarrow \text{VCYao.Verify}_{\text{SK}_{\text{Yao}}}(\sigma_y)$. Note that this step does not require communication. Client C_1 does the same to verify the computation.

3.3 Generalization for multiple clients

Our protocol RMC-PVC can be generalized for any number of clients without modification. Assume $f : \{0,1\}^n \to \{0,1\}^m$ a boolean function where n denotes the number of clients. By C_i with $i \in \{1, \ldots, n\}$ we denote a client. All clients want to compute $f(x_1, \ldots, x_n) = y$ where x_i is an input known by C_i. As stated in the initial setup, we consider that all clients share SK. Each client C_i sends $\{\sigma_{x_i}\}_{pk_f, pk_{x_i}} \leftarrow \text{MKHE.Enc}(pk_{x_i}, \sigma_{x_i}) + \{0\}_{pk_f}$ to the server as explained in the two-client case. The server computes and sends back to every client $\{\sigma_y\}_{pk_f, pk_{x_1}, \ldots, pk_{x_n}}$ using PK. Each client C_i sends $\text{Part}_i(\sigma_y)$ to the server. The server sends to all clients $\{\text{Part}_1(\sigma_y), \ldots, \text{Part}_n(\sigma_y)\}$ the set of partial decryptions. As presented in the two-case presentation, each client computes the partial decryption of the encrypted encoded result $\text{Part}_f(\sigma_y) \leftarrow \text{MKHE.PartDec}(sk_f, \{\sigma_y\}_{pk_f, pk_{x_1}, \ldots, pk_{x_n}})$ (with sk_f obtained from SK). From $\text{Part}_f(\sigma_y)$ and the received set of partial decryptions, each client C_i computes the merge encoded output defined as $\sigma_y \leftarrow \text{MKHE.Merge}(\text{Part}_f(\sigma_y), \text{Part}_1(\sigma_y), \ldots, \text{Part}_n(\sigma_y))$. Each client C_i is able to retrieve y and to verify the computation performed by the server by computing $y \leftarrow \mathsf{VCYao.Verify}_{\mathsf{SK}_{\mathsf{Yao}}}(\sigma_y)$. If the function outputs $\perp$, then the evaluation performed by the server is invalid and therefore rejected by the clients. Otherwise, each client accepts the result y.

4 SECURITY OVERVIEW

Due to the lack of space, we present the security arguments for RMC-PVC. The proofs are presented in the long version[RMC 2022]. We first explain our security model in which we prove the security.

Behavior of clients and server. We consider clients to be *honest-but-curious* (they follow the protocol as expected but try to learn information from the other clients inputs), and the untrusted server to be *malicious* (*i.e.,* does not follow the protocol as expected).

Verification queries. In our protocol, we reuse the *same* garbled circuit, which implies that the output labels are generated once at the beginning and do not change along the different computations. Since we use a homomorphic scheme, the server is able to manipulate the output labels and asks clients for a verification bit (1 if returned computation is valid, 0 otherwise). Given such oracle, the server can recover a bit of a label by XORing a label with an output label. With the response to the verification query of the XOR-ed label, the server can recover the label's bit. In a such situation, a client has been turned into a forgery for the server. Therefore, to address this issue, while the server returns valid computations, clients are allowed to respond with a verification bit. However, when invalid computation is received, verification queries are not allowed anymore. For further details, we refer the reader to [Gennaro et al. 2010] describing the same issue.

Impact of collusion between two clients. Our security model assumes that two or more clients can collude together. This remark is justified by the fact that clients are assumed honest-but-curious. Two colluding clients can be seen as a single client producing two inputs for the function, which does not break any property.

Impact of collusion between a client and the server. In our use-case, since hospitals depends on a trusted authority such as a government, we can legitimately have a common secret between the different hospitals (or clients). In case where a client is corrupted by the server, the common secret is revealed. This scenario breaks the output privacy and the security properties. Each client C_i sends to the server its input x_i encrypted under both pk_i and pk_f. In case where a client $C_{j \neq i}$ is corrupted by the server, revealing sk_f. So, the input-privacy holds only on the privacy of sk_i. However, the server can ask for the partial decryption of $\{x_i\}_{sk_i}$ with sk_i and compute the partial decryption with sk_f by its own, leading to reveal x_i by merging the two partial decryptions. Hence, input-privacy holds while C_i does not compute partial decryption. Hence, clients and the server must not collude, which is a reasonable hypothesis in our use-case where hospitals depend on a trusted authority.

In the long version, we prove the following theorems:

Theorem 4.1 (Correctness of RMC-PVC). *Let* Π *be a correct semantically secure* MKHE *scheme with a distributed decryption property and* Φ *be a Yao-Secure symmetric encryption scheme. Then, for every function* f, RMC-PVC *is correct.*

Proof (sketch). The correctness of RMC-PVC relies on both the correctness of the garbled circuit (constructed with a correct Yao-Secure symmetric encryption scheme) and the correctness of the MKHE scheme Π. By removing first the homomorphic encryption layer, we obtain a garbled circuit execution in clear. As stated in [Lindell and Pinkas 2008; Yao 1986], a garbled circuit is correct. Hence, RMC-PVC is correct.

Theorem 4.2 (Input-Privacy of RMC-PVC). *Let* Π *be a correct semantically secure* MKHE *scheme with a distributed decryption property and* Φ *be a Yao-Secure symmetric encryption scheme. Then, for every function* f, RMC-PVC *is input-private.*

With original garbled circuits, the privacy holds since the input provided by the client to the server is encoded. At each bit b of the input for a wire $w \in \mathcal{W}_{\text{in}}$ corresponds a label k_w^b. The server is not able to distinguish from label k_w^b either if b equals 0 or if b equals 1, since the label is chosen independantly of b. In RMC-PVC, the privacy holds on the same principle, even if the same is used through multiple executions but encrypted under different keys. Therefore, if an adversary $\mathcal{A}$ is able to break the privacy of RMC-PVC, then we can create an adversary $\mathcal{B}$ able to break the privacy of the garbled circuit in game $\mathsf{Exp}_{\mathsf{Priv}}^{\mathcal{B}}$ presented in fig. 3.

Theorem 4.3 (Security of RMC-PVC). *Let* Π *be a correct semantically secure* MKHE *scheme with a distributed decryption property and* Φ *be a Yao-Secure symmetric encryption scheme. Then, for every function* f, RMC-PVC *is secure.*

To produce an invalid result with garbled circuits, the server should return a computation where at least one output label k_w^b as been replaced by k_w^{1-b} for some wire $w \in \mathcal{W}$. Nevertheless, the server has only a negligible probability to recover the label k_w^{1-b}, at the condition that the garbled circuit is executed *only one time*. Otherwise, the next executions could return the other label k_w^{1-b}, letting the server to cheat without being detected. In RMC-PVC, the garbled circuit can be used an arbitrary number of times thanks to the MKHE scheme. At each new execution of the garbled circuit, a fresh MKHE key pair is generated by clients, preventing the server to reuse the labels from previous executions. In the security proofs, we show that if such an adversary $\mathcal{A}$ manage to break the security of our protocol, then we can construct an $\mathcal{B}$ able to break the security of garbled circuit in game $\mathsf{Exp}_{\mathsf{Sec}}^{\mathcal{B}}$ presented in Fig. 2.

5 COMPLEXITY ANALYSIS OF RMC-PVC

We study the asymptotic execution time of RMC-PVC. Let $f = (\mathsf{G}, \mathcal{W}, \mathcal{W}_{\text{in}}, \mathcal{W}_{\text{out}})$ be an arbitrary boolean circuit f where G is the set of gates, $\mathcal{W}$ the set of wires, $\mathcal{W}_{\text{in}}$ the set of input wires and $\mathcal{W}_{\text{out}}$ the set of output wires.

Step (1). During step (1), each client encodes and encrypts his input. We recall the considered function f is expected to be a boolean circuit, having $\mathcal{W}_{\text{in}}$ input bits, the step (1) requires a $O(O(\mathcal{W}_{\text{in}}) \times O(\mathsf{MKHE.Enc}))$ complexity.

Step (2). In this step, the server evaluates the function f using the encrypted garbled circuits $\{\gamma\}_{pk_f}$, over the encrypted encoded inputs $\{\{\sigma_{x_i}\}_{pk_f, pk_{x_i}}\}_n^{i=1}$ sent by clients. The server executes the boolean circuit gate by gate as described in Section 3 until to get $\{\sigma_y\}_{pk_f, pk_{x_1}, \ldots, pk_{x_n}}$. The server evaluates gate after gate from the input to the output of the boolean circuits. At each gate, the server must execute the circuit Γ in order to obtain the label for the next gate. Hence, the server requires a $O(|\mathsf{G}| \times O(\Gamma))$ complexity. We can refine the $O(\Gamma)$ complexity, since Γ tries to decrypt a ciphertext (produced by the Sym scheme) among γ_g containing 2^i inputs where i is the number of inputs of each gate. Since we consider gates with two input wires, then i equals to 2, leading to get four ciphertexts in γ_g. For each ciphertext, Γ executes the decryption algorithm Sym.Dec twice, with the two provided labels. At most, there is four decryptions using Sym.Dec, leading to the complexity of $O(|\mathsf{G}| \times 8O(\mathsf{Sym.Dec}))$.

Step (3). In this step, each client C_i computes $\mathsf{Part}_i(\sigma_y)$ the partial decryption of the ciphertext. Clearly, the complexity of this step depends of the complexity of the MKHE.PartDec function, leading to the complexity $O(\mathsf{MKHE.PartDec})$.

Step (4). During this step, the server is in charge to broadcast the partial decryptions to all clients. Only communications are required in this step: the server receives the partial decryptions provided by each client, creates a set of received partial decryptions then sent to every clients. Therefore, the complexity is constant.

Step (5). Each client starts by computing the partial decryption $\mathsf{Part}_f(\sigma_y)$. Given this last partial decryption, each client computes the result using the function MKHE.Merge, which is verified with VCYao.Verify working in a constant time. Hence, we obtain $O(O(\mathsf{MKHE.PartDec}) + O(\mathsf{MKHE.Merge}) + |\mathcal{W}_{\mathrm{out}}|)$.

Compared to the closest work of this paper [Gennaro et al. 2010], a client performs the same operations to encode his input *i.e.,* $O(|\mathcal{W}_{\mathrm{in}}|) \times O(\mathsf{MKHE.Enc})$. Still on the client side, the result decryption differs from [Gennaro et al. 2010] since we rely on the distributed decryption. In their protocol a client is able to verify the result in $|\mathcal{W}_{\mathrm{out}}|O(\mathsf{FHE.Dec})$ operations whereas we requires $|\mathcal{W}_{\mathrm{out}}|(O(\mathsf{MKHE.PartDec})+O(\mathsf{MKHE.Merge}))$ operations. On the other side, the server requires $|\mathsf{G}|O(\mathsf{Sym.Dec})$ operations for the function evaluation which is stricly less than $|\mathsf{G}|O(\mathsf{Sym.Dec} + \mathsf{FHE.Enc})$ obtained in [Gennaro et al. 2010]. Hence, the function evaluation in our protocol requires $|\mathsf{G}|\mathsf{FHE.Enc}$ operations.

6 CONCLUSION

In RMC-PVC, each client can verify the correctness of the evaluation performed by the server, thanks to the partial decryption property. We proved that RMC-PVC is *correct*. More importantly, we proved that RMC-PVC is *private* in the sense that the server as well as the other clients cannot learn a client's input. RMC-PVC is *secure* meaning that the server cannot produce an invalid result accepted by the cients. Finally, we study the complexity of our protocol and shows that a client outsoucing computations performs less computations compared to the locally computed function case.

In future, we plan to prepare an open-source implementation of this protocol, proving the effectiveness we have claimed in this paper. Moreover, developping a protocol secure even with verification queries and against server-clients collusions will be an intersting way of improvements.

ACKNOWLEDGMENTS

This study was partially supported by the French ANR project MobiS5 ANR-18-CE39-0019, SEVERITAS ANR- 20-CE39-0009, and the Datalake for Nuclear Project funded by France Relance.

REFERENCES

2022. RMC-PVC: A Multi-Client Reusable Verifiable Computation Protocol (Long version). https://eprint.iacr.org/2022/1748.

Alexandre Bois, Ignacio Cascudo, Dario Fiore, and Dongwoo Kim. 2021. Flexible and Efficient Verifiable Computation on Encrypted Data. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12711 LNCS (2021), 528–558. https://doi.org/10.1007/978-3-030-75248-4_19/COVER

Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. 2019. Efficient Multi-Key Homomorphic Encryption with Packed Ciphertexts with Application to Oblivious Neural Network Inference. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 395–412. https://doi.org/10.1145/3319535.3363207

Long Chen, Zhenfeng Zhang, and Xueqing Wang. 2017. Batched Multi-hop Multi-key FHE from Ring-LWE with Compact Ciphertext Extension. In *Theory of Cryptography*, Yael Kalai and Leonid Reyzin (Eds.). Springer International Publishing, Cham, 597–627.

Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Carlos Cid. 2013. Multi-Client Non-interactive Verifiable Computation. In *Theory of Cryptography*, Amit Sahai (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 499–518.

Dario Fiore, Anca Nitulescu, and David Pointcheval. 2020. Boosting Verifiable Computation on Encrypted Data. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12111 LNCS (2020), 124–154. https://doi.org/10.1007/978-3-030-45388-6_5

Rosario Gennaro, Craig Gentry, and Bryan Parno. 2010. Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. In *Advances in Cryptology – CRYPTO 2010*, Tal Rabin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 465–482.

Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing* (Bethesda, MD, USA) *(STOC '09)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/1536414.1536440

Shafi Goldwasser, S. Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. 2014. Multi-input Functional Encryption. In *33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques* (33rd annual international conference on the theory and applications of cryptographic techniques ed.). Springer Berlin Heidelberg, 578–602. https://www.microsoft.com/en-us/research/publication/multi-input-functional-encryption/

Shafi Goldwasser, Yael Tauman Kalai, and Raluca Ada Popa. 2013. Reusable Garbled Circuits and Succinct Functional Encryption. In *In Proceedings of the 45th annual ACM symposium on Theory of computing (STOC)* (in proceedings of the 45th annual acm symposium on theory of computing (stoc) ed.). ACM. https://www.microsoft.com/en-us/research/publication/reusable-garbled-circuits-succinct-functional-encryption/

Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. 2015. Delegating Computation: Interactive Proofs for Muggles. *J. ACM* 62, 4, Article 27 (sep 2015), 64 pages. https://doi.org/10.1145/2699436

Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. 2014. Amortizing garbled circuits. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8617 LNCS. Issue PART 2. https://doi.org/10.1007/978-3-662-44381-1_26

Seny Kamara, Payman Mohassel, and Mariana Raykova. 2011. Outsourcing Multi-Party Computation. *IACR Cryptol. ePrint Arch.* 2011 (2011), 272.

Pascal Lafourcade, Gael Marcadet, and Léo Robert. 2022. Faster Non-interactive Verifiable Computing. Cryptology ePrint Archive, Paper 2022/646. https://eprint.iacr.org/2022/646 https://eprint.iacr.org/2022/646.

Yehuda Lindell and Benny Pinkas. 2008. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology* 22 (2008), 161–188.

Yehuda Lindell and Benny Pinkas. 2015. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. *Journal of Cryptology* 28 (2015). Issue 2. https://doi.org/10.1007/s00145-014-9177-x

Silvio Micali. 2000. Computationally Sound Proofs. *SIAM J. Comput.* 30, 4 (oct 2000), 1253–1298. https://doi.org/10.1137/S0097539795284959

Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. 162–167. https://doi.org/10.1109/SFCS.1986.25