THC: Practical and Cost-Effective Verification of Delegated Computation

Pablo Rauzy and Ali Nehme

Université Paris 8, Saint-Denis, France
pr@up8.edu — ali.link@hotmail.com

Abstract. Homomorphic cryptography is used when computations are delegated to an untrusted third-party. However, there is a discrepancy between the untrustworthiness of the third-party and the silent assumption that it will perform the expected computations on the encrypted data. This may raise serious privacy concerns, for example when homomorphic cryptography is used to outsource resource-greedy computations on personal data (e.g., from an IoT device to the cloud). In this paper we show how to cost-effectively verify that the delegated computation corresponds to the expected sequence of operations, thus drastically reducing the necessary level of trust in the third-party. Our approach is based on the well-known *modular extension* scheme: it is transparent for the third-party and it is not tied to a particular homomorphic cryptosystem nor depends on newly introduced (and thus less-studied) cryptographic constructions. We provide a proof-of-concept implementation, THC (for *trustable homomorphic computation*), which we use to perform security and performance analyses. We then demonstrate its practical usability, in the case of a toy electronic voting system.

1 Introduction

Delegating computation to a third-party is pretty common nowadays, with the proliferation of small devices like smartphones and tablets which are mostly terminal interfaces for cloud services. This tendency is even accelerating with the so-called *Internet of Things*. Indeed, a lot of low-power and low-performance devices are now getting connected together and, most of the time, to centralized and proprietary cloud services. Most of these devices are supposedly made to make people's life better, but part of the process is the monitoring of personal user data, for example smartwatches may collect the location and the level of physical activities of their wearer.

Hence, serious privacy concerns need to be addressed. When data only need to be stored or transmitted from the device to the cloud or from a user to another, classical cryptography (symmetric and asymmetric) can solve the problem. However, most of the time users' data have to be processed, e.g., to generate statistics or to compute quantities that are more informational than the raw values collected by the devices. Homomorphic cryptography allows users to encrypt their data before they are sent to the cloud for further processing. Computations can

then be performed on the encrypted values, and the result can be sent back to the users, who are able to decrypt it.

While this may seem to be enough to solve the privacy issue (depending on the definition of privacy), it is not enough for users to be able to fully trust the third-party performing the computation on their homomorphically encrypted data. Indeed, there is no reason to trust the third-party with the execution of the expected sequence of operations.

For the sake of simplicity, consider this dummy example: an insurance company offers multiple options (at different prices) to their clients depending (among other things) on how well they want to be covered for weight-related diseases. In order to help their clients to choose the option that better suit their need, the insurance company offers a service that watches their body mass index ($\text{BMI} = \frac{\text{mass}}{\text{height}^2}$) over time. However, people do not want their private data such as mass and height to be sent in clear over the network, *nor to be revealed to their insurance company*. This is where homomorphic cryptography can help. Here, the user's device would send $\mathcal{E}(\text{mass})$ and $\mathcal{E}(\text{height})$ the homomorphically encrypted values of the user' mass and height to the insurance's cloud service, which would perform the BMI computation on the encrypted values and return it to the user, who would decrypt it and use the information to decide which insurance plan to choose. In this scenario, the user does not trust the insurance company with their personal data, but we still assume that the BMI computation is performed correctly, i.e., *the insurance company is trusted with the computation even if it is not considered trustworthy*. Yet, it is in the interest of the insurance company to sell their more expensive plans, so maybe their service would instead compute $\text{BMI}' = \frac{\mathcal{E}(\text{mass})+20}{\mathcal{E}(\text{height})^2}$ to influence the user's choice...

From this dummy example, we understand that in order to be able to delegate computation to an untrusted third-party, we need to have a way to verify the integrity of the delegated computation results[1].

Related works. There are existing works on the subject [14,13,9,10]. However, these attempts at verifiable delegation of homomorphic computation are either impractical and/or introduce complex cryptographic constructions and rely on them. They also require the collaboration of the untrusted third-party, which our method does not. For example, the work of Lai et al. [10], which is the closest to what we want to achieve ourselves, introduces a new cryptographic primitive called "homomorphic encrypted authenticator", and stays at a theoretical level (it does not provide an implementation). The lack of practical and usable implementations of related work to benchmark THC against (for both security and performance) is a real concern. We believe our implementation[2] is an important contribution in this regard.

[1] Note that in a real-world IoT situation dealing with e.g., complex health or position data, it is important for the verification to be cost-effective.

[2] THC is available at `https://code.up8.edu/pablo/thc`. It can also be installed directly with `pip3 install thc`.

Contributions. In this paper we present THC, a method to practically and cost-effectively implement *trustable homomorphic computation*. Our goal is to provide a way to verify the integrity of delegated computations generically, in order to let as much freedom as possible in the choice of the homomorphic cryptosystem to use. We provide a proof-of-concept implementation[2] in Python that we use to analyze the security and performance of the proposed method. For demonstration purpose, we also used our THC implementation to build an electronic voting system that lets a group of agents organize a secret vote using an untrusted third-party server.

Organization of the paper. In the next section, we detail modular extension, the technique on which THC is based. After that, we present our proof-of-concept implementation in Section 3. We then use it to study the security and performance of the proposed method respectively in Section 4 and Section 5. In Section 6, we demonstrate THC in a practical use case by building an electronic voting system. Finally, we draw conclusions and think of perspectives in Section 7.

2 Modular Extension

Our goal is to be able to verify the integrity of a computation $\mathcal{C}$ performed on our behalf by an untrusted third-party. Of course, the verification has to be inexpensive compared to carrying out the computation $\mathcal{C}$ by ourselves.

There is another area of cryptography where the same problem with the same constraint exists: implementation security against physical attacks. One kind of physical attacks consists in injecting fault during the cryptographic computation (e.g., using an electromagnetic impulse targeted at the processor performing the computation) in the hope that the intermediate values that are tampered with (e.g., the content of a register that gets randomized) will influence the final result of the computation in such a way that will help breaking the cryptography. Such attacks have been demonstrated to be feasible since 1997 when Boneh *et al.* presented the BellCoRe[3] attack [3] which essentially reduces the complexity of retrieving an RSA private key to computing a gcd instead of solving an integer factorization problem.

In 1999, Shamir presented a countermeasure [18] to the BellCoRe attack. Shamir's idea is based on the principle of *modular extension* (see Fig. 1). It consists in lifting the computation into an over-structure (e.g., an overring $\mathbb{Z}_{pr}$) which allows to quotient the result of the computation back to the original structure (e.g., $\mathbb{F}_p$), as well as quotienting a "checksum" of the computation to a smaller structure (e.g., $\mathbb{F}_r$). What has just been described is the *direct product* of the underlying algebraic structures. If an equivalent computation is performed in parallel in the smaller structure, its result can be compared with the checksum of the main computation. If they are the same, we have a high confidence in the integrity of the main computation.

[3] The attack is named after the *Bell Communication Research* labs, where it was discovered.

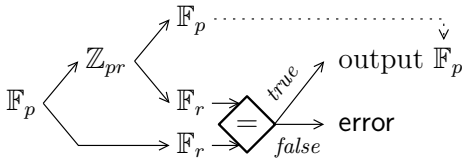

Fig. 1: Sketch of the principle of *modular extension*.

Although it was originally designed to protect CRT-RSA[4], the modular extension scheme has been successfully ported to elliptic curve scalar multiplication [2,1], and has been formally studied in both settings [15,16,6]: the cost of the countermeasure is minimal, and the non-detection probability is provably inversely proportional to the security parameter r (the size of the small structure).

Getting back to our concerns, a nice property of the modular extension scheme that, to the best of our knowledge, has not been taken advantage of yet, is that the "small computation" over $\mathbb{F}_r$ can be carried out independently from the "big computation" over $\mathbb{Z}_{pr}$, and can thus be performed on another machine entirely.

Hence our main idea: leveraging modular extension to verify the integrity of delegated homomorphic computations. By doing so, we rely on a well-established tried-and-tested method rather than introducing novel cryptographic construction that would still have to withstand the test of time. Implementation is straightforward: delegate the "big computation" over $\mathbb{Z}_{pr}$, locally perform the "small computation" over $\mathbb{F}_r$, compare the results modulo r and either return the verified result of the delegated computation modulo p, or signal an error, according to the modular extension scheme.

3 THC Implementation

In this section, we present our proof-of-concept implementation of THC. We have multiple goals with this implementation:

- show that THC is *generic*: the implementation should be able to work with any homomorphic cryptosystem as long as its ciphertexts live in a modular structure such as a field or a ring;
- show that THC is *secure*: the probability of not detecting an error in the delegated computation is inversely proportional to the security parameter;
- show that THC is *cost-effective*: the verification of delegated computations using THC should be nearly free;
- show that THC is *practical*: it should be easy to use in a realistic system.

[4]CRT-RSA is an optimization of RSA using the Chinese Remainder Theorem, which makes it vulnerable to the BellCoRe attack but is indispensable on low-end devices such as credit cards (it provides an almost 4× speed-up and allows for security parameters 2× bigger).

3.1 The Core

The core of the implementation consists of three classes: the `THC` implementation itself, and two interfaces[5]: `HomomorphicCryptosystem` and `Computation`. We will first present these interfaces and then the `THC` class.

`HomomorphicCryptosystem`. This interface requires five methods:
- a constructor, to setup the cryptosystem;
- `encrypt`, which takes a plaintext as argument and returns a ciphertext;
- `decrypt`, which does the opposite;
- `get_modulus`, which returns the characteristic of the ring in which the ciphertexts live; and
- `mod`, which applies a modulus to a ciphertext[6].

`Computation`. This interface requires two methods:
- `local`, which takes a modulus and an array of arguments, performs a computation with them and returns the result modulo the given modulus; and
- `remote`, which is supposed to query the untrusted third-party to perform the same computation.

`THC`. This class is where the modular extension scheme is implemented. Apart from its constructor, where it is given an `HomomorphicCryptosystem` instance $\mathcal{H}$ a `Computation` instance $\mathcal{C}$, and the security parameter r, it has two methods:
- `compute`, which takes a list of arguments $\langle a_i \rangle$ and
  1. encrypts its arguments:
     $\langle c_i \rangle \leftarrow \mathcal{H}.\texttt{encrypt}(\langle a_i \rangle)$,
  2. does the remote computation in $\mathbb{Z}_{Nr}$ (where N is obtained using the $\mathcal{H}.\texttt{get\_modulus()}$):
     $\texttt{result}_{Nr} \leftarrow \mathcal{C}.\texttt{remote}(Nr, \langle c_i \rangle)$,
  3. does the local computation in $\mathbb{F}_r$:
     $\texttt{result}_r \leftarrow \mathcal{C}.\texttt{local}(r, \langle c_i \bmod r \rangle)$,
  4. verifies the result using the second method:
- `verify`, which takes $\texttt{result}_{Nr}$ and $\texttt{result}_r$ as arguments and, as per the modular extension scheme presented in Section 2: compares $\texttt{result}_{Nr} \bmod r$ and $\texttt{result}_r$ for equality, then returns $\mathcal{H}.\texttt{decrypt}(\texttt{result}_{Nr} \bmod N)$ if the comparison succeeds, or returns $\perp$ (i.e., `False`) otherwise.

A trivial example. To explain more clearly how everything interacts and how THC is used, here is an example of `HomomorphicCryptosystem` implementation, which does not actually do any encryption[7], followed by an example of `Computation` implementation, which allows to instantiate linear polynomials:

[5] We use the `abc` Python lib (see `https://docs.python.org/3/library/abc.html`) for that, as Python object-model does not natively support abstract classes or interfaces.

[6] This is necessary because some cryptosystems have ciphertexts that are not plain numbers, e.g., ElGamal uses pairs and the modulus needs to be applied to both elements independently.

[7] Remark that it is quite homomorphic nonetheless ;).

```
class Field (HomomorphicCryptosystem):
    def __init__ (self, p):
        self._p = p
    def get_modulus (self):
        return self._p
    def encrypt (self, m):
        return m % self._p
    def decrypt (self, c):
        return c % self._p
    def mod (self, c, mod):
        return c % mod

class Linear (Computation):
    def __init__ (self, a, b):
        self.a = a
        self.b = b
    def local (self, mod, args):
        return (self.a * args[0] + self.b) % mod
    def remote (self, mod, args):
        c = Cloud.PolynomialAPI() # imaginary
        c.compute_in_ring(mod)
        return c.linear([self.a, self.b], args[0])
```

Given these implementations, the `THC` class could be used like this, with the security parameter $r = 17$:

```
>>> thc = THC(Field(p=59233), Linear(42, 51), 17)
>>> y = thc.compute([2021])
```

Here, `y` should be $(42 \times 2021 + 51) \bmod 59233$, that is 25700. According to what was explained in Section 2, THC will call the `Linear.remote` method with 59233×17 as `mod` and the "encrypted" value of 2021 in `args`, while the `Linear.local` method is called with 17 as `mod` and the "encrypted" value of 2021 modulo 17 (i.e., 15) in `args`.

We know that the local computation will return $(42 \times 15 + 51) \bmod 17 = 1$. If our imaginary Cloud Polynomial API gives a wrong result, its comparison with the local computation modulo 17 will most likely fail[8], and THC will return `False`. Otherwise the server returns the expected 84933 which satisfies $84933 \equiv 1 \bmod 17$, so THC will return the results modulo 59233 to give the expected answer: 25700.

3.2 Homomorphic Cryptosystems

For a given homomorphic cryptosystem to work with THC, the condition is that its ciphertexts live in a field or ring structure. This constraint still leaves a lot of choices on the table. To demonstrate the genericity of THC, we chose to implement four different homomorphic cryptosystems (in addition to the trivial one we already presented), mostly chosen for their simplicity of implementation: RSA, ElGamal, Paillier, and HE1. For illustration purpose, we will present the first two in details, including their implementation code.

RSA. The first cryptosystem we implemented is RSA [17], which is homomorphic for multiplications when used without padding (which should really never be the case when any level of security is required). The implementation of textbook RSA is quite straightforward (`modinv` is the modular inverse):

[8] In real settings, much bigger security parameters are used.

```
class RSA (HomomorphicCryptosystem):
    def __init__ (self, p, q, e):
        self._N = p * q
        self._e = e
        self._d = modinv(e, (p - 1) * (q - 1))
    def get_modulus (self):
        return self._N
    def encrypt (self, m):
        return pow(m, self._e, self._N)
    def decrypt (self, c):
        return pow(c, self._d, self._N)
```

We do not need to implement the `mod` method as the trivial version is actually already provided by the `HomomorphicCryptosystem` base class. We can test RSA's homomorphic property in a Python interpreter where `rsa` is a properly instantiated `RSA` object:

```
>>> c1, c2 = rsa.encrypt(43), rsa.encrypt(47)
>>> rsa.decrypt(c1 * c2)
2021 # 43 * 47 is 2021
```

ElGamal. The ElGamal cryptosystem [8] is also homomorphic for multiplications. In addition to be able to multiply ciphertext, ElGamal can also do scalar multiplication homomorphically thanks to its malleability. Its implementation is more complex than that of RSA, but this is an occasion to show that an `HomomorphicCryptosystem` implementation can consist of glue code that calls an existing external cryptographic library (namely PyCryptodome[9]):

```
import Crypto.PublicKey.ElGamal as EG
from Crypto.Random.random import StrongRandom as SR
class ElGamal (HomomorphicCryptosystem):
    def __init__ (self, p, g, y, x):
        self._p = p
        self._eg = EG.construct((p, g, y, x))
    def get_modulus (self):
        return self._p
    def encrypt (self, m):
        r = SR.randint(1, self._p - 1)
        return self._eg.encrypt(m, r)
    def decrypt (self, c):
        return self._eg.decrypt(c)
    def mod (self, c, mod):
        return (c[0] % mod, c[1] % mod)
```

Since ElGamal ciphertexts are pairs, we also have an example of how the `mod` method is used. Again, given a properly instantiated `ElGamal` object `elg` in a Python interpreter, we can test the homomorphic properties of ElGamal:

```
>>> def scalar_mul (c, k):
...     return (c[0], c[1] * k)
>>> def mul (a, b):
...     return (a[0] * b[0], a[1] * c[1])
>>> c1, c2 = elg.encrypt(43), elg.encrypt(47)
>>> elg.decrypt(mul(c1, c2))
2021 # the value of 43 * 47 hasn't changed
>>> elg.decrypt(scalar_mul(c1, 10))
430
```

[9] Python Cryptography Toolkit, `https://www.pycryptodome.org/`.

Paillier. The Paillier cryptosystem [12] can do homomorphic additions (the product of ciphertexts corresponds to the addition of plaintexts) and thus scalar multiplication. We will use it in Section 6 to build an electronic voting system.

```
>>> def add (a, b):
...     return a * b
>>> def scalar_mul (c, k):
...     return c ** k
>>> ten = pai.encrypt(10)
>>> three = pai.encrypt(3)
>>> pai.decrypt(add(ten, three))
13
>>> pai.decrypt(scalar_mul(ten, 3))
30
```

HE1. In their 2017 paper [7], Dyer et al. present several variants of their homomorphic encryption over the integers. From the initially proposed version HE1, they derive variants allowing to mitigate brute force guessing attack even if the inputs distribution has insufficient entropy, and then variants where they add dimensions to the ciphertexts to enhance security. For our testing purpose, we chose to implement the initial version presented in the paper, as it is easy to code and yet allows for homomorphic additions and multiplications at the same time, enabling to compute any polynomials homomorphically.

```
>>> a, b = he1.encrypt(11), he1.encrypt(3)
>>> c, d = he1.encrypt(2), he1.encrypt(9)
>>> he1.decrypt(a * b + c * d)
51
```

4 Security Analysis of THC

Before looking at experimental data, we start with some theoretical background. In the following, we call *delegated computation* the one we ask the third-party to perform, and *remote computation* the one the third-party actually performs.

4.1 Theoretical background

The goal of THC is to verify the integrity of a delegated computation. As such, its level of security can be defined as its probability of detecting that the remote computation has been tampered with. This probability is equal to $1-\mathbb{P}_{\text{nd}}$, where $\mathbb{P}_{\text{nd}}$ is the probability of non-detection.

The formal study of $\mathbb{P}_{\text{nd}}$ is carried out in the third section of Dugardin et al's paper [6] in the case of single and multiple faults in the computation. The precise value of $\mathbb{P}_{\text{nd}}$ depends on the specific computation that is delegated, but it is shown that $\mathbb{P}_{\text{nd}} \approxeq \frac{1}{r}$ where r is the chosen security parameter. In the appendices of the same paper the authors predict a theoretical upper-bound of $\frac{57}{r}$ in practical cases. We will not repeat the full proof here but we are still going to sketch it below.

Computations that concern us are polynomials (i.e., we can do additions and multiplications) of the input variables (the ciphertexts that we send to the third-party). We call $P(x_1, x_2, \ldots, x_n)$ the polynomial corresponding to the delegated computation, where the x_i are the input variables. We give the formal name P' to the remote computation, i.e., the computation of P that might have been

tampered with. The polynomials P and P' can differ on everything: constant terms, like terms, and degree.

Let N be the modulus of the homomorphic cryptosystem we are using. Let r be our security parameter. Let $c_1, c_2, \ldots, c_n$ be the ciphertexts that we send to the third-party we delegate the computation P to. THC will detect an error if $P'(c_1, c_2, \ldots, c_n) \not\equiv P(c_1, c_2, \ldots, c_n) \bmod r$. That is, errors will not be detected if and only if: $P'(c_1, c_2, \ldots, c_n) \equiv P(c_1, c_2, \ldots, c_n) \bmod r$, while $P'(c_1, c_2, \ldots, c_n) \not\equiv P(c_1, c_2, \ldots, c_n) \bmod N$. Note that if the second condition is not fulfilled, there is actually no errors in the result of the computation. Hence, $\mathbb{P}_{\text{nd}}$ is the probability of having $P(c_1, c_2, \ldots, c_n) - P'(c_1, c_2, \ldots, c_n) \equiv 0 \mod r$. We call ΔP the polynomial $P - P'$. For random tampering with the remote computation, we have $\mathbb{P}_{\text{nd}} = \frac{\#\text{roots}(\Delta P)}{r}$, where #roots gives the number of roots of a given polynomial in $\mathbb{F}_r$.

One could argue that a malicious third-party may not *randomly* tamper with our computation. However, we recall that due to the modular reduction by r, the inputs and coefficients of ΔP are effectively randomized. Thus, it is actually reasonable to consider ΔP's coefficients and its inputs to be random and evenly distributed over $\mathbb{F}_r$, which allows us to conclude that **the probability of non-detection of errors $\mathbb{P}_{\text{nd}}$ is indeed inversely proportional to the security parameter r.**

Moreover, in his 2006 paper [11], Leont'ev demonstrates that "*the number of zeros of a random polynomial lying inside the field $\mathbb{F}_q$ has, asymptotically as $q \to \infty$, a Poisson distribution with parameter $\lambda = 1$. In particular, a random polynomial over $\mathbb{F}_q$ has "on the average", as q increases, exactly one root in $\mathbb{F}_q$*". This means that the proportionality constant #roots(ΔP) of $\mathbb{P}_{\text{nd}}$ is such that $\mathbb{P}_{\text{nd}} \approx \frac{1}{r}$. In particular, when r is a random 32-bit prime number as it would be the case in most practical situations, we have that $\mathbb{P}_{\text{nd}} \approx 10^{-9}$.

Nevertheless, a malicious third-party could retrieve the value of r, e.g., because in some cases we might need to provide it with the value of Nr, so that the result of the computation of P does not grow too large, and they obtain r from that by factorizing Nr. In practice, N is either prime (e.g., in ElGamal) or the product of two big primes (e.g., in RSA, Paillier, and HE1), and r is prime, but potentially much smaller, typically it is a randomly chosen 32-bit prime number, so obtaining r this way is realistic.

In such cases, a malicious third-party could easily tamper with our computation in an undetected manner. However, it would be limited to adding multiples of r to the final result of the computation. Indeed, only multiples of r can be added in P' to bypass THC's verification. Moreover, we recall that this tampering happens on homomorphically encrypted values. This means that there is no way for a malicious third-party to perform a precise attack on the actual (i.e., decrypted) result of the computation without breaking the homomorphic encryption scheme. In practice, a malicious third-party is thus limited to vandalism. In cases where this vandalism is not obvious once the results are decrypted (e.g., a proposition getting billions of billions of votes when there are only a few dozens of participants in the vote), it is still possible to detect it by delegating

the same computation twice using different values for r. If both computation do not return the same seemingly valid result, vandalism is detected. Note that this strategy requires to retrieve the result of the delegated computation a first time before delegating the same computation again using a different value for r. Otherwise, the malicious third-party can tamper with both computations in the same way by adding a multiple of the product of the two r values (which is admittedly even more constraining).

Remark that with extremely malleable homomorphic cryptosystems like HE1 which supports scalar additions and multiplications (besides additions and multiplications between ciphertexts), the decrypted result of an erroneous remote computation will keep the property of being the actual result of the computation added to a multiple of r, i.e., $\exists k \in \mathbb{Z}$ such that $\mathcal{D}(P'(c_1, \ldots, c_n)) = \mathcal{D}(P(c_1, \ldots, c_n)) + kr$, where $\mathcal{D}$ is our homomorphic decryption function. In such situations, if r is chosen large enough to be bigger than the biggest expected result, it is possible to detect errors that bypassed THC's verification and even to get the correct result back by reducing the erroneous result modulo r.

4.2 Experimental study

For our experimental study of THC's security, we wrote several implementations of `Computation` specifically for testing and analysis purposes[10]. These implementations' `remote` method do not actually query a third-party, but rather perform the same computation as the `local` method does, except that in order to simulate a misbehaving third-party, it inserts a random fault in the computation. Three of these have been used to test the probability of non-detection:

- `Product` computes the product of its arguments (for RSA, Paillier, HE1);
- `PairProduct` does the same thing but element-wise on pairs (for ElGamal);
- `RandomBinaryPolynomial` takes a degree at instantiation and generates a random binary polynomial of this degree, that we used with HE1.

Each of these computations were tested with random numbers and using randomly chosen r on 2, 4, 8, 16, and 32 bits.

The results are presented in Table 1a. In all cases, the number of missed error quickly drops as r size increases until it reaches a satisfying 0 when r is a 32-bit prime. However, we remark that for small r sizes, only the configuration where the HE1 homomorphic cryptosystem is used to perform a `RandomBinaryPolynomial` computation (called `HE1-poly` in the table) corresponds to the theoretically predicted probability of non-detection (i.e., $\mathbb{P}_{\text{nd}} \approx \frac{1}{r}$, as per Section 4.1). This is actually not so surprising. Indeed, Leont'ev's result [11] concerns the number of roots for a *random* polynomial, which is exactly what we have in HE1–poly. Moreover, Leont'ev's result is valid in $\mathbb{F}_r$ when $r \to \infty$, so it is expected that it does not hold for very small r. These experimental results confirm the predicted influence of the security parameter.

[10] Scripts used to produce and analyze our experimental data are available in our Python package repository at `https://code.up8.edu/pablo/thc`.

Table 1: Experimental results.

Experiment	r **size** (bits)	**#runs**	**#missed**	**ratio**
RSA–prod	2	1000	810	0.81
	4	1000	269	0.269
	8	1000	30	0.03
	16	50000	6	0.00012
	32	**100000**	**0**	**0.0**
ElGamal–prod	2	1000	897	0.897
	4	1000	372	0.372
	8	1000	56	0.056
	16	50000	8	0.00016
	32	**100000**	**0**	**0.0**
Paillier–prod	2	1000	896	0.896
	4	1000	395	0.395
	8	1000	19	0.019
	16	50000	6	0.00012
	32	**100000**	**0**	**0.0**
HE1–prod	2	1000	904	0.904
	4	1000	383	0.383
	8	1000	31	0.031
	16	50000	7	0.00014
	32	**100000**	**0**	**0.0**
HE1–poly	2	1000	376	0.376
	4	1000	57	0.057
	8	1000	4	0.004
	16	50000	1	0.00005
	32	**100000**	**0**	**0.0**

(a) Experimentally observed security of THC.

	Mean time (µs)			
Experiment	in $\mathbb{Z}_N$	in $\mathbb{Z}_{Nr}$	in $\mathbb{F}_r$	**cost**
RSA–prod	77.144 ± 2.54	79.036 ± 2.65	0.485 ± 0.03	3.08%
	76.575 ± 2.34	78.550 ± 2.27	0.495 ± 0.04	3.22%
	76.465 ± 2.72	78.570 ± 2.42	0.561 ± 0.05	3.49%
	76.580 ± 2.96	79.052 ± 3.03	0.655 ± 0.06	4.08%
	77.382 ± 3.49	79.349 ± 3.26	0.752 ± 0.07	3.51%
ElGamal–prod	107.400 ± 5.09	108.705 ± 4.74	1.257 ± 0.14	2.39%
	107.320 ± 2.49	107.686 ± 2.77	1.236 ± 0.09	1.49%
	107.885 ± 3.74	108.948 ± 3.63	1.416 ± 0.18	2.30%
	107.666 ± 4.24	108.709 ± 4.64	1.563 ± 0.20	2.42%
	109.061 ± 6.07	110.778 ± 6.13	2.164 ± 0.28	3.56%
Paillier–prod	308.094 ± 68.99	307.876 ± 67.74	0.519 ± 0.12	0.10%
	268.897 ± 16.11	269.238 ± 15.29	0.466 ± 0.05	0.30%
	269.308 ± 8.96	269.641 ± 8.44	0.533 ± 0.07	0.32%
	270.559 ± 12.48	271.879 ± 12.02	0.608 ± 0.08	0.71%
	271.324 ± 13.06	272.549 ± 13.66	0.717 ± 0.10	0.72%
HE1–prod	117.082 ± 20.80	117.053 ± 20.85	0.661 ± 0.12	0.54%
	114.053 ± 22.38	114.033 ± 22.80	0.660 ± 0.13	0.56%
	118.745 ± 20.47	118.617 ± 20.33	0.773 ± 0.14	0.54%
	122.356 ± 16.44	122.459 ± 16.88	0.923 ± 0.13	0.84%
	120.954 ± 17.10	121.104 ± 17.20	1.066 ± 0.15	1.01%
HE1–poly	520.025 ± 16.46	518.606 ± 17.36	3.538 ± 0.25	0.41%
	515.870 ± 4.64	516.368 ± 5.57	3.666 ± 0.20	0.81%
	517.472 ± 9.81	516.168 ± 11.18	4.137 ± 0.29	0.55%
	532.335 ± 35.14	531.020 ± 35.19	5.140 ± 0.70	0.72%
	531.379 ± 25.29	531.682 ± 23.52	6.345 ± 0.82	1.25%

(b) Experimentally observed time-cost of THC.

5 Performance Analysis of THC

Using THC costs both in additional memory usage and computation time. The additional memory usage strongly depends on the specific application, but is precisely predictable. Indeed, for each homomorphically encrypted number, THC needs to keep its residue modulo r, so the additional memory used correspond to the number of sensible information that are needed for the delegated computation multiplied by the size of r.

The additional computation time also depends on r. It is possible to get a sense of it by observing random computations, using the same settings as in Section 4.2, except that the "remote" computation does not insert errors[10].

The results are presented in Table 1b. For all configurations, N is a 2048-bit number. Times are expressed in microseconds and are means computed over 1000×1000 runs, that is 1000 random computations that are executed 1000 times each in a loop to get something measurable and averaged. Each mean is accompanied by its standard deviation. These computations were executed on an Intel® Core™ i5-6300U CPU @ 2.40GHz, with Python version 3.7.3.

The *cost* is defined as the additional time it takes to compute in $\mathbb{Z}_{Nr}$ instead of $\mathbb{Z}_N$ added to the time it takes to perform the local computation in $\mathbb{F}_r$. It is expressed in percentage of the computation time in $\mathbb{Z}_N$.

Note that in most applications, the local computation in $\mathbb{F}_r$ can actually be performed in parallel to the remote computation in $\mathbb{Z}_{Nr}$, meaning the time cost we present is largely overestimated. Indeed, as can be seen in Table 1b, most of the cost comes from the local computation in $\mathbb{F}_r$. This can be explained by the fact that the size of N (2048 bits) and Nr (between 2050 and 2080 bits) are very similar. So much in fact that the standard deviation of the experimental cost is sometimes bigger than their difference.

These results show that **the cost of verifying a delegated computation using THC is minimal**. Indeed, the cost in additional computation time is on the order of a percent of the unverified computation time, and the cost in additional memory usage is linear with regard to the number of inputs of the delegated computation, with a constant factor depending on the size of the security parameter (typically 4 bytes). This is particularly encouraging, as computations delegated on homomorphically encrypted data tend to be expensive, owing to the size of the ciphertexts.

6 Use Case: Electronic Voting

In this section we sketch an example of THC use case[11] to help readers get a better grasp of both its utility and usability. We start by presenting a scenario, then a "naive" solution without THC, then we show how to use THC to improve over this solution. Some explanations are doubled with Python code for clarity. Note that although the presented code was written for this sole purpose and is thus extremely simplified, it should still work as expected when executed.

[11] The demo voting client and server are included in the THC Python package available at `https://code.up8.edu/pablo/thc`.

Scenario. A group of agents wants to organize a vote to make some decisions. It can be a group of people (e.g., an association or a political group) or it can be some kind of sensor network that need to make a centralized decision. These agents have secure means of communication between them so they can securely exchange information or share a common secret. However, the vote has to be secret in order to avoid influence bias in the case of a group of people, or to avoid privacy concerns with regard to the data collected by the sensors (which may be owned by different people).

To keep each vote secret for all participants, votes are cast to a third-party. This third-party is an untrusted service provider, typically a voting plateform in the cloud. It should not be made aware of any participant's choice, and is not trusted with the counting of votes either. In addition, participants should be able to verify that others votes are valid (no cheating).

We will use homomorphic cryptography to ensure that the voting plateform can store all the votes and count the result without being able to snoop on any participant's vote nor on the final result. We will use our proposed *trustable homomorphic computation* scheme to ensure the validity of each vote as well as the integrity of the vote count.

We will see how to implement a *cumulative voting* system, where each participant has a given number of points and freely assign them to each proposition (or candidates). Cumulative voting is frequently used in federated organizations to take decision at a federal level, as each of the federated groups usually has a number of votes that depends non-linearly on its size. We remark that *plurality voting* is a particular case of cumulative voting where each participant has a single point and thus can vote for a single proposition.

Note that a *score voting* system, where each participant gives a score chosen among a finite number of possibilities (e.g., an integer between 0 and 10) to each proposition, can be implemented as an overlay on cumulative voting: it is equivalent to have a plurality vote between the possible scores for each propositions. Also remark that *approval voting* is a particular case of score voting where the score is either 0 or 1.

Threat model. The third-party (1) should not be able to learn about any of the participants' vote, (2) should not be able to learn about the result of the votes, and (3) should not be able to manipulate the votes. Participants can securely exchange the secret keys that have to be kept secret from the third-party.

Please note that we are not actually trying to build a production-ready electronic voting system and that we disregard a lot of security details that would be mandatory for such an application (e.g., making sure it is not possible to vote twice, or to vote on behalf of someone else without their consent, etc.).

Initial solution. We need to be able to count votes, so a natural choice is to use the Paillier cryptosystem, which can perform additions homomorphically, to encrypt the votes. We call $\mathcal{P}_{\text{enc}}$ and $\mathcal{P}_{\text{dec}}$ the encryption and decryption functions of the Paillier cryptosystem. We have that $\mathcal{P}_{\text{dec}}(\mathcal{P}_{\text{enc}}(m_1) \times \mathcal{P}_{\text{enc}}(m_2)) = m_1 + m_2$ (for easier reading, we omit the private and public key arguments to

these functions). We also use a symmetric encryption scheme $(\mathcal{E}, \mathcal{D})$, such that $\mathcal{D}(\mathcal{E}(m)) = m$ (again, we omit the secret key).

We call $p_1, \ldots, p_n$ the propositions for a given vote. A ballot is a pair of the form $((b_1, \ldots, b_n), id)$ where b_i corresponds to the number of votes for p_i, and id uniquely identifies the agent of which it is the ballot. The sum $\sum_{i=1}^{n} b_i$ must be less than or equal to the number of votes V_{id} that the agent id has. For example, if there are 3 propositions, both $([0, 0, 7], a)$ and $([1, 2, 3], a)$ are valid ballots for a if a has 7 votes, but $([3, 4, 5], a)$, $([3, 4], a)$, or $([2, 2, 2, 1], a)$ are not. The encrypted version of the ballot (b, id) that is sent to the third-party is the pair $((c_1, \ldots, c_n), hid)$ where $c_i = \mathcal{P}_{\text{enc}}(b_i)$ and $hid = \mathcal{E}(id)$.

For each ballot (c, hid), the third-party can compute the product $C_{hid} = \prod_{i=1}^{n} c_i$ and make it public so that each participant can verify that $\mathcal{P}_{\text{dec}}(C_{hid})$ is at most equal to the number of votes that the agents identified by $id = \mathcal{D}(hid)$ is supposed to have.

Given the list of all submitted ballots $\mathcal{B}$, the encrypted results of the vote $(r_1, r_2, \ldots, r_n)$ can be computed by the third-party such that $r_i = \prod_{j=1}^{\#\mathcal{B}} \mathcal{B}_{j,i}$, where $\mathcal{B}_{j,i}$ is the c_i of the jth ballot in $\mathcal{B}$. The participants can retrieve the results of the vote $(P_1, \ldots, P_n)$ by computing each $P_i = \mathcal{P}_{\text{dec}}(r_i)$.

Problem. The presented solution is almost sufficient: threats (1) and (2) are covered by the Paillier cryptosystem. However, threat (3) is still a problem. Consider the following scenario. A group of people organize a secret vote on the platform provided by the third-party T. The question is "Should we move away from T to organize our votes?", the propositions are "yes" and "no", and each participant has one vote. In practice, people who want to chose another voting platform vote "yes", i.e. $(1, 0)$, those who want to continue using T vote "no", i.e., $(0, 1)$, and those who do not care about the platform either submit a blank vote, i.e., $(0, 0)$, or do not participate at all.

```
from thc.crypto.paillier import Paillier
from thc.utils import prime
paillier = Paillier(prime(1024), prime(1024))
mod = paillier.get_modulus()
ballots = []
def cast_vote (choice):
    if choice == 'yes':
        y, n = paillier.encrypt(1), paillier.encrypt(0)
    elif choice == 'no':
        y, n = paillier.encrypt(0), paillier.encrypt(1)
    else:
        y, n = paillier.encrypt(0), paillier.encrypt(0)
    ballots.append((y, n))
```

Initially, most people do not really care about the voting platform and many do not participate in the vote. At some point one person decides to dig into the subject and discovers that T makes use of analytics and advertisements trackers on their web interface. This person then decides to loudly campaign in favor of the "yes", explaining why using trackers is wrong and how it violates users privacy. Soon enough, more people participate in the vote.

```
cast_vote('yes'), cast_vote('no'), cast_vote('blank')
cast_vote('no'), cast_vote('yes'), cast_vote('no')
# here the campaign for the "yes" happens
```

```
cast_vote('yes'), cast_vote('yes'), cast_vote('no')
cast_vote('yes'), cast_vote('blank'), cast_vote('yes')
```

Of course it is not possible to know if the campaign convinced them or merely reminded them about the vote, before it is counted. When the vote is closed the platform should compute the result on the encrypted data and return it.

```
from functools import reduce
def result (votes, m):
    return reduce(lambda a, b: (a * b) % m, votes)
res_y = result([b[0] for b in ballots], mod)
res_n = result([b[1] for b in ballots], mod)
```

Then, the participants can retrieve the results and decrypt them.

```
yes, no = paillier.decrypt(res_y), paillier.decrypt(res_n)
```

In our example, `yes` is 6 and `no` is 4: the group decided to move away from T. However, T heard about the campaign against them and because T is clearly an evil company, they decide to manipulate the vote to have a better chance of keeping their users. Instead of taking the all the vote into account as they are, T replaces all the ballots with copies of ballots randomly chosen among those they received before the campaign against them happened.

```
from random import randint
for i in range(len(ballots)):
    forged_ballots.append(ballots[randint(0,5)])
res_y = result([b[0] for b in forged_ballots], mod)
res_n = result([b[1] for b in forged_ballots], mod)
```

The number of votes stays the same, the C_{hid} for each ballot seems okay (i.e., it is either 0 or 1 when decrypted). For the participants, there are no particular reasons to suspect a manipulation of the vote. Indeed, T cannot know the vote of any specific person nor the final result thanks to the homomorphic encryption. However, at least if the campaign for the "yes" was convincing enough, their manipulation probably biased the results in their favor.

This time, when decrypted by the participants, `yes` could be 4 and `no` could be 7, for example, thereby changing the vote result and the group decision in favor of T. These results, while manipulated, look totally legit: as explained before, the campaign for the "yes" may have reminded people to vote without convincing them.

Using THC. In a parallel universe, the same events happen, except the participants of the vote decide to use THC to verify the integrity of the vote count. At the beginning they choose a small random prime r that they keep along the other secrets (i.e., the two big primes used for Paillier), then they adapt the modulus given to T, and they create an instance of `THC`[12].

```
r = prime(32)                                   # new
mod = paillier.get_modulus() * r              # modified
thc = THC(paillier, None, r)                    # new
```

They keep track of the residue modulo r of the votes. These are shared among participants so that any one of them can verify the results.

[12]Here we will only use the `verify` method of THC, so a `Computation` instance is not necessary. See Section 3.1.

```
ballots_r = []                                      # new
def cast_vote (choice):
    # ...
    ballots_r.append((y % r, n % r))                # new
```

When the vote is closed, participants compute the results modulo r on their side.

```
res_y_r = result([b[0] for b in ballots_r], r)  # new
res_n_r = result([b[1] for b in ballots_r], r)  # new
```

And when they receive the results from T, they use THC to verify the integrity of the delegated vote count.

```
yes, no = thc.verify(res_y, res_y_r), thc.verify(res_n, res_n_r)  # modified
```

This time, if T manipulates the vote, `yes` and `no` will be `False`, otherwise they will contain the actual result of the vote: threat (3) is no longer a problem.

Note that we do not dwell on it here, but THC could also be used to venify the integrity of each C_{hid} if thought necessary. More importantly, **remark the practical usability of THC**: it only requires a few lines of code on the client side, and is entirely transparent on the server side.

More problems. Using THC, we significantly reduced the necessary trust in third-parties to which computations are delegated. However, we are still far from having removed the necessity of the trust entirely: THC can only verify the integrity of computational aspects of the delegated data processing. Logical aspects are left unverified (e.g., in our scenario, if T publishes the list of ballots, the secrecy of the vote is broken as any participants can decrypt the ballots[13]).

7 Conclusions and Perspectives

In this paper, we presented a method for verifying the integrity of delegated computations, targeted in particular at delegated computations on homomorphically encrypted data. We provide an implementation of this method called THC (for *trustable homomorphic computation*) that we used to assess the genericity, the security, and the cost of the method. We also detailed a practical use case.

We showed both in theory and in practice that THC is secure, cost-effective, and practically usable. Our implementation itself, or any implementation of the modular extension, can be used in existing code at minimal cost both in terms of development and run-time resources, thereby reducing the necessary trust in third-parties to which computations on sensible data are delegated (e.g., cloud service providers).

Nonetheless, we did not achieve the goal of not having to trust the third-party *at all*. Indeed, logical aspects that must be ensured by the third-party cannot be verified using THC. In the scenario we develop in Section 6 for example, if the voting platform publishes the encrypted vote count incrementally after each vote rather than only when the vote is closed, it becomes possible to break the vote secrecy for anyone who knows when someone else voted.

[13] Again, we never intended to design a production-ready eletronic voting system This particular problem could be mitigated using the right cryptographic tools, but is still relevant to illustrate our point here.

A state-of-the-art implementation of homomorphic encryption, TFHE [5], has been used by the CEA-LIST crypto team to build Cingulata [4], a compiler that translate arbitrary C++ programs into Boolean circuits that are homomorphically evaluated using TFHE. With Cingulata, the logic of the program is protected by design, as it is embedded into the homomorphically encrypted circuit to be evaluated by a third-party. However, the third-party could still mess with the evaluation. Since TFHE ciphertexts live on a torus, which should share the properties necessary for modular extension, it would be interesting to study the feasibility of using the THC method to verify the integrity of the evaluation of delegated Cingulata circuits.

References

1. Baek, Y.J., Vasyltsov, I.: How to Prevent DPA and Fault Attack in a Unified Way for ECC Scalar Multiplication - Ring Extension Method. In: Information Security Practice and Experience (2007), https://sci-hub.tw/10.1007/978-3-540-72163-5_18
2. Blömer, J., Otto, M., Seifert, J.P.: Sign Change Fault Attacks on Elliptic Curve Cryptosystems. In: Fault Diagnosis and Tolerance in Cryptography (2006), https://eprint.iacr.org/2004/227
3. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults. In: EuroCrypt (1997), https://link.springer.com/content/pdf/10.1007%2F3-540-69053-0_4.pdf
4. CEA-LIST Crypto Team: Cingulata: a compiler toolchain and RTE for running C++ programs over encrypted data by means of fully homomorphic encryption techniques (2018), https://github.com/CEA-LIST/Cingulata
5. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: Fast Fully Homomorphic Encryption Library (2016), https://tfhe.github.io/tfhe/
6. Dugardin, M., Guilley, S., Moreau, M., Najm, Z., Rauzy, P.: Using modular extension to provably protect edwards curves against fault attacks. Cryptology ePrint Archive, Report 2015/882 (2015), https://eprint.iacr.org/2015/882
7. Dyer, J., Dyer, M.E., Xu, J.: Practical Homomorphic Encryption Over the Integers. International Journal of Information Security (2017), https://arxiv.org/abs/1702.07588
8. ElGamal, T.: A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In: Advances in Cryptology (1985), https://link.springer.com/content/pdf/10.1007%2F3-540-39568-7_2.pdf
9. Fiore, D., Gennaro, R., Pastro, V.: Efficiently verifiable computation on encrypted data. Cryptology ePrint Archive, Report 2014/202 (2014), https://eprint.iacr.org/2014/202
10. Lai, J., Deng, R., Pang, H., Weng, J.: Verifiable computation on outsourced encrypted data. In: Computer Security - ESORICS 2014 (2014), https://link.springer.com/content/pdf/10.1007%2F978-3-319-11203-9_16.pdf
11. Leont'ev, V.: Roots of random polynomials over a finite field. Mathematical Notes **80**(1-2) (2006), https://sci-hub.tw/10.1007/s11006-006-0139-y
12. Paillier, P.: Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In: Advances in Cryptology — EUROCRYPT '99 (1999), https://link.springer.com/content/pdf/10.1007%2F3-540-48910-X_16.pdf

13. Parno, B., Gentry, C., Howell, J., Raykova, M.: Pinocchio: Nearly practical verifiable computation. Cryptology ePrint Archive, Report 2013/279 (2013), https://eprint.iacr.org/2013/279
14. Parno, B., Raykova, M., Vaikuntanathan, V.: How to delegate and verify in public: Verifiable computation from attribute-based encryption. Cryptology ePrint Archive, Report 2011/597 (2011), https://eprint.iacr.org/2011/597
15. Rauzy, P., Guilley, S.: A Formal Proof of Countermeasures Against Fault Injection Attacks on CRT-RSA. Journal of Cryptographic Engineering (2014), https://eprint.iacr.org/2013/506
16. Rauzy, P., Guilley, S.: Countermeasures Against High-Order Fault-Injection Attacks on CRT-RSA. In: IACR Workshop on Fault Diagnosis and Tolerance in Cryptography (2014), https://eprint.iacr.org/2014/559
17. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM (1978), https://people.csail.mit.edu/rivest/Rsapaper.pdf
18. Shamir, A.: Method and apparatus for protecting public key schemes from timing and fault attacks (1999), US Patent Number 5,991,415 (https://www.google.com/patents/US5991415)