A multivariate noise-free HE proposal

Gerald Gavin[1] and Sandrine Tainturier[2]

[1] Laboratory ERIC - University of Lyon
gerald.gavin@univ-lyon1.fr
[2] Adecco - Geneve
sandrine-tainturier@orange.fr

Abstract. In [Gav16], [GB19], [GT20], new ideas to build homomorphic encryption schemes have been presented. Authors propose private-key encryption schemes whose secret key is a rational function ϕ/ϕ'. By construction, these schemes are not homomorphic. To get homomorphic properties, nonlinear homomorphic operators are derived from the secret key. In [GT20], an additive homomorphic encryption is proposed. In this paper, we adopt the same approach to build a HE based on the same private-key encryption scheme. We obtain a multivariate encryption scheme in the sense that the knowledge of the CPA attacker can be turned into an over-defined system of nonlinear equations (contrarily to LWE-based encryptions). The factoring assumption is introduced in order to make a large class of attacks based on Grœbner basis irrelevant. While we did not propose a formal security proof relying on a classical cryptographic assumption, we hopefully provide convincing evidence for security.

1 Introduction

The prospect of outsourcing an increasing amount of data storage and management to cloud services raises many new privacy concerns for individuals and businesses alike. The privacy concerns can be satisfactorily addressed if users encrypt the data they send to the cloud. If the encryption scheme is homomorphic, the cloud can still perform meaningful computations on the data, even though it is encrypted.

The theoretical problem of constructing a fully homomorphic encryption scheme (HE) supporting arbitrary functions f, was only recently solved by the breakthrough work of Gentry [Gen09]. More recently, further fully homomorphic schemes were presented [SS10],[vDGHV10],[CNT12],[GHS12a],[GSW13] following Gentry's framework. The underlying tool behind all these schemes is the use of Euclidean lattices, which have previously proved powerful for devising many cryptographic primitives. A central aspect of Gentry's fully homomorphic scheme (and the subsequent schemes) is the ciphertext refreshing Recrypt operation. Even if many improvements have been made in one decade, this operation remains very costly [LNV11], [GHS12b], [DM15], [CGGI18]. Indeed, bootstrapped bit operations are still about one billion times slower than their plaintext equivalents (see [CGGI18]).

We adopt a recent approach developed in [Gav16], [GB19], [GT20] where the secret key is a (multivariate) rational function ϕ_D/ϕ'_D. A ciphertext is here a randomly chosen vector $\boldsymbol{c}$ over $\mathbb{Z}_n$ satisfying $\phi_D/\phi'_D(\boldsymbol{c}) = x$. In particular, an encryption $\boldsymbol{c}$ of 0 satisfied $\phi_D(\boldsymbol{c}) = 0$. It follows that the expanded representations of ϕ_D should not be polynomial-size (otherwise the CPA attacker could recover it by solving a polynomial-size linear system). In order to get polynomial-time encryptions and decryptions, ϕ_D/ϕ'_D should be written in a compact form, e.g. a factored or semi-factored form. By construction, the generic cryptosystem described above is not homomorphic in the sense that the vector sum is not a homomorphic operator. To get homomorphic properties, we develop *ad hoc* nonlinear homomorphic operators Add and Mult, sometimes denoted by $\oplus$ or $\otimes$.

1.1 Overview of the paper

We consider a private-key encryption scheme where the secret key is a randomly chosen square $2\kappa - by - 2\kappa$ matrix S defined over $\mathbb{Z}_n$, n being a RSA modulus. To encrypt x, randomly choose $x_1, \ldots, x_\kappa, r_1, \ldots, r_\kappa$ in $\mathbb{Z}_n$ s.t. $x_1 + \ldots + x_\kappa = x$ and output

$$\boldsymbol{c} = S^{-1} \begin{pmatrix} r_1 x_1 \\ r_1 \\ \cdots \\ r_\kappa x_\kappa \\ r_\kappa \end{pmatrix}$$

Throughout this paper, we will use the following convenient notation:

$$X(\boldsymbol{c}) \stackrel{\mathsf{def}}{=} (x_1, \ldots, x_\kappa)$$
$$R(\boldsymbol{c}) \stackrel{\mathsf{def}}{=} (r_1, \ldots, r_\kappa)$$

Clearly, $\boldsymbol{c}$ is an encryption of 0 if and only if

$$\phi_D(\boldsymbol{c}) \stackrel{\mathsf{def}}{=} \sum_{\ell=1}^{\kappa} \langle \boldsymbol{s}_{2\ell-1}, \boldsymbol{c} \rangle \prod_{\ell' \neq \ell} \langle \boldsymbol{s}_{2\ell'}, \boldsymbol{c} \rangle = 0$$

where $\boldsymbol{s}_i$ refers to the i^{th} row of S.

In Section 2, we propose some security results based on symmetry under the factoring assumption. In particular, these results will ensure that the secret key S cannot be recovered under the factoring assumption.

In Section 3, we formally present the private-key encryption scheme described above. The basic attack of this scheme consists of recovering the monomial coefficients of ϕ_D by solving a linear system. The key underlying idea is that the expanded representation of ϕ_D is exponential-size (and thus cannot be recovered) provided

$$\kappa = \Theta(\lambda)$$

In Section 4, we propose simple/naive homomorphic operators keeping some symmetry properties. The (naive) operator Add exactly follows the one considered in [GT20]. This operator is nonlinear (quadratic) and ensures that

$$X(\boldsymbol{c} \oplus \boldsymbol{c}') = (x_1 + x'_1, \ldots, x_\kappa + x'_\kappa)$$
$$R(\boldsymbol{c} \oplus \boldsymbol{c}') = (r_1 r'_1, \ldots, r_\kappa r'_\kappa)$$

A high-level description of this operator is proposed in Figure 1.

$$\mathsf{Add}\left(S^{-1}\begin{pmatrix} r_1 x_1 \\ r_1 \\ \cdots \\ r_\kappa x_\kappa \\ r_\kappa \end{pmatrix}, S^{-1}\begin{pmatrix} r'_1 x'_1 \\ r'_1 \\ \cdots \\ r'_\kappa x'_\kappa \\ r'_\kappa \end{pmatrix}\right) = S^{-1}\begin{pmatrix} r_1 r'_1 (x_1 + x'_1) \\ r_1 r'_1 \\ \cdots \\ r_\kappa r'_\kappa (x_\kappa + x'_\kappa) \\ r_\kappa r'_\kappa \end{pmatrix}$$

Fig. 1. Description of the naive operator Add.

The implementation of Mult is a little bit more complex. It cannot be achieved by applying only one quadratic operator. Indeed, it exploits the equality

$$xx' = \sum_{i=1}^{\kappa} \sum_{j=1}^{\kappa} x_i x'_j$$

It follows that at least κ operators are necessary to *store* all the products $x_i x'_j$ in some intermediate vectors. In the naive implementation of Mult, this is achieved by applying κ quadratic operators $\mathcal{O}_1, \ldots, \mathcal{O}_\kappa$. For instance, a high-level description of the naive operator $\mathcal{O}_1$ is given in Figure 2: $\mathcal{O}_1(\boldsymbol{c}, \boldsymbol{c}')$ outputs an encryption of $x_1 x'_1 + \ldots + x_\kappa x'_\kappa$.

$$\mathcal{O}_1\left(S^{-1}\begin{pmatrix} r_1 x_1 \\ r_1 \\ \cdots \\ r_\kappa x_\kappa \\ r_\kappa \end{pmatrix}, S^{-1}\begin{pmatrix} r'_1 x'_1 \\ r'_1 \\ \cdots \\ r'_\kappa x'_\kappa \\ r'_\kappa \end{pmatrix}\right) = S^{-1}\begin{pmatrix} r_1 r'_1 x_1 x'_1 \\ r_1 r'_1 \\ \cdots \\ r_\kappa r'_\kappa x_\kappa x'_\kappa \\ r_\kappa r'_\kappa \end{pmatrix}$$

Fig. 2. Description of the naive operator $\mathcal{O}_1$.

It then suffices to homomorphically add these vectors, i.e.

$$\mathsf{Mult}(\boldsymbol{c}, \boldsymbol{c}') = \mathcal{O}_1(\boldsymbol{c}, \boldsymbol{c}') \oplus \cdots \oplus \mathcal{O}_\kappa(\boldsymbol{c}, \boldsymbol{c}')$$

A security analysis is proposed in Section 5. We first prove a fundamental result based on symmetry assuming the hardness of factoring. Proposition 3 states

that relevant information about S cannot be recovered. Roughly speaking, only symmetric values (evaluations of polynomials over S) can be efficiently recovered, while natural compact representations of ϕ_D do not deal with symmetric values. This result excludes a large class of attacks based on Grœbner basis or more generally based on variable elimination theory.

We then exhibit some weaknesses of this scheme by listing some efficient attacks. We mainly adopt this approach in a pedagogical point of view. We wish to provide some intuition about the ideas *behind* our construction. In particular, we exhibit a common point of all the identified attacks. This leads us to propose an (informal) assumption (see Assumption 1). This will then guide us to develop new homomorphic operators in Section 6. These operators can be seen as a randomization/generalization of the naive ones.

While Assumption 1 was not formally reduced to the security of our scheme, we modestly think that some strong evidence about this are provided.

Remark 1. A SageMath implementation of the HE is given in Appendix B. The source code of this implementation and the sources of some attacks proposed in this paper can be found in the following archive:

https://drive.google.com/drive/folders/1fkma-sacLO5LA7eqgln-D7OTpYXN6xCQ?usp=sharing

1.2 Notation

We use standard Landau notations. Throughout this paper, we let λ denote the security parameter: all known attacks against the cryptographic scheme under scope should require $2^{\Omega(\lambda)}$ bit operations to mount. Let $\kappa \geq 2$ be an integer and let n be a large prime or a RSA modulus. All the computations considered in this paper will be done in $\mathbb{Z}_n$.

- *Δ_κ refers to the set of permutations over $\{1, \ldots, \kappa\}$.*
- *$\Sigma_\kappa = \{\sigma_1, \ldots, \sigma_\kappa\} \subset \Delta_\kappa$ defined by $\sigma_i(j) = (i + j - 2 \bmod \kappa) + 1$, , i.e. $\sigma_i(1) = i; \sigma_i(2) = i + 1; \ldots; \sigma_i(\kappa) = i - 1$.*
- *The cardinality of a set S will be denoted by $\#S$.*
- *'*Choose at random $x \in X$*' will systematically mean that x is chosen according to uniform probability distribution over X, i.e. $x \xleftarrow{\$} X$.*
- *The inner product of two vectors $\boldsymbol{v}$ and $\boldsymbol{v}'$ is denoted by $\langle \boldsymbol{v}, \boldsymbol{v}' \rangle$*
- *The set of all square $t - by - t$ matrices over $\mathbb{Z}_n$ is denoted by $\mathbb{Z}_n^{t \times t}$.*

Remark 2. The number $M(m, d)$ of m-variate monomials of degree d is equal to $\binom{d+m-1}{d}$. In particular, $M(2\kappa, \kappa) \approx (27/4)^\kappa$.

2 Some security results under the factoring assumption

Throughout this section, n denotes a randomly chosen RSA-modulus. Given a function $\phi : \mathbb{Z}_n^r \to \mathbb{Z}_n$, $z_\phi \stackrel{\mathsf{def}}{=} \#\{x \in \mathbb{Z}_n^r | \phi(x) = 0\}/n^r$. Classically a polynomial will be said *null* (or identically null) if each coefficient of its expanded representation is equal to 0.

2.1 Roots of polynomials

The following result proved in [AM09] establishes that it is difficult to output a polynomial ϕ such that z_ϕ is non-negligible. The security of RSA in the generic ring model can be quite straightforwardly derived from this result (see [AM09]).

Theorem 1. ***(Lemma 4 of [AM09] and Proposition 1 of [GT20]).*** *Assuming factoring is hard, there is no p.p.t algorithm $\mathcal{A}$ which inputs n and which outputs*[1] *a $\{+, -, \times\}$-circuit representing a non-null polynomial $\phi \in \mathbb{Z}_n[X_1, \ldots, X_r]$ such that z_ϕ is non-negligible.*

Thanks to this lemma, showing that two polynomials (*built without knowing the factorization of n*) are equal with non-negligible probability becomes an algebraic problem: it suffices to prove that they are identically equal.

2.2 Symmetry

Let $\kappa \geq 2$ and $t \geq 1$ be integers. Recall that Δ_κ denotes the set of the permutations over $\{1, \ldots, \kappa\}$. Throughout this section, we will consider an arbitrary subset $\Sigma \subseteq \Delta_\kappa$. Let y_1, y_2 be randomly chosen in $\mathbb{Z}_n$. It is well-known that recovering y_1 with non-negligible probability given only $S = y_1 + y_2$ or $P = y_1 y_2$ is difficult assuming the hardness of factoring (y_1, y_2 are the roots of the polynomial $y^2 - Sy + P$). In this section, we propose to extend this. The following definition naturally extends the classical definition of symmetric polynomials.

Definition 1. *Consider the tuples of indeterminate $(Y_\ell = (X_{\ell 1}, \ldots, X_{\ell t}))_{\ell=1,\ldots,\kappa}$. A polynomial $\phi \in \mathbb{Z}_n[Y_1, \ldots, Y_\kappa]$ is Σ-symmetric if for any permutation $\sigma \in \Sigma$,*

$$\phi(Y_1, \ldots, Y_\kappa) = \phi(Y_{\sigma(1)}, \ldots, Y_{\sigma(\kappa)})$$

Let $\mathcal{P}$ be an arbitrary p.p.t algorithm which inputs n and outputs m Σ-symmetric polynomials $s_1, \ldots, s_m$ and a non Σ-symmetric polynomial π. Evaluating π only given evaluations of $s_1, \ldots, s_m$ is difficult.

Lemma 1. *Let n be a randomly chosen RSA modulus and $(s_1, \ldots, s_m, \pi) \leftarrow \mathcal{P}(n)$. Assuming the hardness of factoring, there is no p.p.t algorithm which outputs $\pi(y)$ given only $s_1(y), \ldots, s_m(y)$ with non-negligible probability over the choice of $n, y \stackrel{\$}{\leftarrow} \mathbb{Z}_n^{\kappa t}$.*

Proof. See [GT20].
□

[1] with non-negligible probability (the coin toss being the choice of n and the internal randomness of $\mathcal{A}$)

3 A private-key encryption scheme

We propose here a private-key encryption scheme where the secret key K contains 2κ randomly chosen secret vectors $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{2\kappa}$ belonging to $\mathbb{Z}_n^{2\kappa}$. Encrypting $x \in \mathbb{Z}_n$ simply consists of randomly choosing $\boldsymbol{c} \in \mathbb{Z}_n^{2\kappa}$ satisfying

$$\frac{\langle \boldsymbol{s}_1, \boldsymbol{c} \rangle}{\langle \boldsymbol{s}_2, \boldsymbol{c} \rangle} + \cdots + \frac{\langle \boldsymbol{s}_{2\kappa-1}, \boldsymbol{c} \rangle}{\langle \boldsymbol{s}_{2\kappa}, \boldsymbol{c} \rangle} = x \tag{1}$$

By assuming the vectors $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{2\kappa}$ linearly independent, our scheme can be defined as follows:

Definition 2. *Let λ be a security parameter. The functions KeyGen, Encrypt, Decrypt are defined as follows:*

- *KeyGen(λ). Let κ be a positive integer indexed by λ, let n be a randomly chosen RSA-modulus. Choose at random an invertible matrix $S \in \mathbb{Z}_n^{2\kappa \times 2\kappa}$. Output*
$$K = \{S\} \ ; \ pp = \{n, \kappa\}$$
- *Encrypt$(K, pp, x \in \mathbb{Z}_n)$. Choose at random $r_1, \ldots, r_\kappa$ in $\mathbb{Z}_n^*$ and $x_1, \ldots, x_\kappa$ in $\mathbb{Z}_n$ s.t. $x_1 + \cdots + x_\kappa = x$. Output*
$$\boldsymbol{c} = S^{-1} \begin{pmatrix} r_1 x_1 \\ r_1 \\ \cdots \\ r_\kappa x_\kappa \\ r_\kappa \end{pmatrix}$$
- *Decrypt$(K, pp, \boldsymbol{c} \in \mathbb{Z}_n^{2\kappa})$. Output*
$$x = \sum_{\ell=1}^{\kappa} \frac{\langle \boldsymbol{s}_{2\ell-1}, \boldsymbol{c} \rangle}{\langle \boldsymbol{s}_{2\ell}, \boldsymbol{c} \rangle}$$
where $\boldsymbol{s}_i$ refers to the i^{th} row of S.

Throughout this paper, $\mathcal{L}_i$ denotes the linear function defined by $\mathcal{L}_i(\boldsymbol{v}) = \langle \boldsymbol{s}_i, \boldsymbol{v} \rangle$. Moreover, $pp = \{n, \kappa\}$ will be assumed to be public. The homomorphic operator(s), developed later, will be included in pp. Proving correctness is straightforward by using the relation $x = r_1 x_1 / r_1 + \ldots + r_\kappa x_\kappa / r_\kappa$. The function Decrypt can be represented by the ratio of two degree-κ polynomials $\phi_D, \phi'_D \in \mathbb{Z}_n[X_1, \cdots, X_{2\kappa}]$ defined by

$$\phi_D = \sum_{\ell=1}^{\kappa} \mathcal{L}_{2\ell-1} \prod_{\ell' \neq \ell} \mathcal{L}_{2\ell'} \ ; \ \phi'_D = \prod_{\ell=1}^{\kappa} \mathcal{L}_{2\ell} \tag{2}$$

i.e.

$$\mathsf{Decrypt}(K, pp, \boldsymbol{c}) = \phi_D(\boldsymbol{c}) / \phi'_D(\boldsymbol{c})$$

At this step, our scheme is not homomorphic in the sense that the vector sum is not an homomorphic operator. Indeed, $\boldsymbol{c}$ and $a \cdot \boldsymbol{c}$ are encryptions of the same message for any $a \in \mathbb{Z}_n^*$.

We can identify two independent sources of randomness in $\mathsf{Encrypt}$: the choice of the *shares* x_i and the choice of the *masks* r_i. As mentioned in the introduction, we will consider the following convenient notation capturing these two types of randomness:

$$X(\boldsymbol{c}) \stackrel{\mathsf{def}}{=} (x_1, \ldots, x_\kappa) = (\mathcal{L}_1(\boldsymbol{c})/\mathcal{L}_2(\boldsymbol{c}), \ldots, \mathcal{L}_{2\kappa-1}(\boldsymbol{c})/\mathcal{L}_{2\kappa}(\boldsymbol{c}))$$
$$R(\boldsymbol{c}) \stackrel{\mathsf{def}}{=} (r_1, \ldots, r_\kappa) = (\mathcal{L}_2(\boldsymbol{c}), \ldots, \mathcal{L}_{2\kappa}(\boldsymbol{c}))$$

The factoring assumption. The factorization of n is not used in KeyGen. Consequently, the generation of n could be externalized[2] (for instance generated by an oracle). In other words, n could be a public input of KeyGen. It follows that all the polynomials considered in our security analysis are built without using the factorization of n. It follows that Proposition 1 and Lemma 1 can be invoked.

The basic attack. The most natural attack consists of solving a linear system in order to recover ϕ_D. Let $\boldsymbol{c} \leftarrow \mathsf{Encrypt}(K, pp, 0)$ be an encryption of 0. By definition, ϕ_D (see (2)) satisfies

$$\phi_D(\boldsymbol{c}) = \prod_{\ell=1}^{\kappa} r_\ell \cdot \sum_{\ell'=1}^{\kappa} x_{\ell'} = 0$$

ensuring that $\phi_D(\boldsymbol{c}) = 0$. By considering several encryptions $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_t$ of 0, we get the system of equations $\phi_D(\boldsymbol{c}_1) = 0, \ldots, \phi_D(\boldsymbol{c}_t) = 0$.

The expanded representation of ϕ_D could be thus recovered[3] by solving a linear system whose variables are its monomial coefficients. However, this attack fails provided $\kappa = \Theta(\lambda)$ because the expanded representation of ϕ_D is exponential-size in this case (see Remark 2). For instance, by choosing $\kappa = 13$, the attack consists of solving a linear system dealing with approximatively $5 \cdot 10^9$ variables: this is currently assumed to be infeasible.

4 A naive implementation of the homomorphic operators

Let $S \leftarrow \mathsf{KeyGen}(\lambda)$. In this section, we will consider the quadratic polynomials $\mathcal{L}_{ij} \in \mathbb{Z}_n[U_1, \ldots, U_{2\kappa}, V_1, \ldots, V_{2\kappa}]$ defined by $\mathcal{L}_{ij}(\boldsymbol{u}, \boldsymbol{v}) = \mathcal{L}_i(\boldsymbol{u})\mathcal{L}_j(\boldsymbol{v}) = \langle \boldsymbol{s}_i, \boldsymbol{u} \rangle \langle \boldsymbol{s}_j, \boldsymbol{v} \rangle$.

In this section, we propose a natural and simple way to implement the homomorphic operators. Throughout this paper, we will consider the following encryptions

[2] ensuring that its factorization was forgotten just after its generation

[3] up to a multiplicative factor

$\boldsymbol{c}$ and $\boldsymbol{c}'$ of x and x'

$$\boldsymbol{c} = S^{-1} \begin{pmatrix} r_1 x_1 \\ r_1 \\ \cdots \\ r_\kappa x_\kappa \\ r_\kappa \end{pmatrix} \quad ; \quad \boldsymbol{c}' = S^{-1} \begin{pmatrix} r'_1 x'_1 \\ r'_1 \\ \cdots \\ r'_\kappa x'_\kappa \\ r'_\kappa \end{pmatrix}$$

4.1 The additive operator

The additive operator exploits the basic equality

$$\frac{a}{b} + \frac{a'}{b'} = \frac{ab' + a'b}{bb'}$$

showing that the numerator and the denominator of the sum can be obtained by evaluating quadratic polynomials over a, a', b, b'. This is the starting point to build an additive operator Add (denoted sometimes $\oplus$) satisfying (see Figure 1)

$$\begin{aligned} X(\boldsymbol{c} \oplus \boldsymbol{c}') &= (x_1 + x'_1, \ldots, x_\kappa + x'_\kappa) \\ R(\boldsymbol{c} \oplus \boldsymbol{c}') &= (r_1 r'_1, \ldots, r_\kappa r'_\kappa) \end{aligned}$$

Definition 3. *$\mathsf{AddGen}(S)$ outputs the expanded representation of the quadratic polynomials $q_1, \ldots, q_{2\kappa}$ defined by*

$$\begin{pmatrix} q_1 \\ \cdots \\ q_{2\kappa} \end{pmatrix} = S^{-1} \begin{pmatrix} \mathcal{L}_{12} + \mathcal{L}_{21} \\ \mathcal{L}_{22} \\ \cdots \\ \mathcal{L}_{2\kappa-1,2\kappa} + \mathcal{L}_{2\kappa,2\kappa-1} \\ \mathcal{L}_{2\kappa,2\kappa} \end{pmatrix}$$

As each quadratic polynomial q_i has $O(\kappa^2)$ monomials, the running time of AddGen is $O(\kappa^4)$ (2κ sums of 2κ quadratic polynomials). The operator $\mathsf{Add} \leftarrow \mathsf{AddGen}(S)$ consists of evaluating the polynomials $q_1, \ldots, q_{2\kappa}$, i.e. $\mathsf{Add}(\boldsymbol{u}, \boldsymbol{v}) = (q_1(\boldsymbol{u}, \boldsymbol{v}), \ldots, q_{2\kappa}(\boldsymbol{u}, \boldsymbol{v}))$, leading to a running time in $O(\kappa^3)$. See Appendix A for a toy implementation of Add.

Proposition 1. *$\mathsf{Add} \leftarrow \mathsf{AddGen}(S)$ is a valid additive homomorphic operator.*

Proof. By construction, for any $\ell = 1, \ldots, \kappa$

$$\begin{aligned} &\langle \boldsymbol{s}_{2\ell-1}, (q_1(\boldsymbol{c}, \boldsymbol{c}'), \ldots, q_{2\kappa}(\boldsymbol{c}, \boldsymbol{c}')) \rangle \\ &= \mathcal{L}_{2\ell-1,2\ell}(\boldsymbol{c}, \boldsymbol{c}') + \mathcal{L}_{2\ell,2\ell-1}(\boldsymbol{c}, \boldsymbol{c}') \\ &= r_\ell x_\ell r'_\ell + r'_\ell x'_\ell r_\ell \\ &= r_\ell r'_\ell (x_\ell + x'_\ell) \end{aligned}$$

and

$$\langle \boldsymbol{s}_{2\ell}, (q_1(\boldsymbol{c}, \boldsymbol{c}'), \ldots, q_{2\kappa}(\boldsymbol{c}, \boldsymbol{c}')) \rangle$$

$= \mathcal{L}_{2\ell,2\ell}(\boldsymbol{c}, \boldsymbol{c}')$
$= r_\ell r'_\ell$

It follows that

$\mathsf{Decrypt}(K, pp, \mathsf{Add}(\boldsymbol{c}, \boldsymbol{c}'))$
$= \frac{r_1 r'_1 (x_1 + x'_1)}{r_1 r'_1} + \ldots \frac{r_\kappa r'_\kappa (x_\kappa + x'_\kappa)}{r_\kappa r'_\kappa}$
$= x_1 + x'_1 + \ldots + x_\kappa + x'_\kappa = x + x'$
□

4.2 The multiplicative operator

The idea behind the operator $\otimes$ exploits the equality

$$xx' = \sum_{1 \leq i,j \leq \kappa} x_i x'_j = \sum_{i=1}^{\kappa} \pi_i$$

with $\pi_i = \sum_{j=1}^{\kappa} x_j x'_{\sigma_i(j)}$ (recall that σ_i is the permutation over $\{1, \ldots, \kappa\}$ defined by $\sigma_i(1) = i$, $\sigma_i(2) = i + 1, \ldots, \sigma_i(\kappa) = i - 1$).

Simply speaking, κ quadratic operators $\mathcal{O}_1, \ldots, \mathcal{O}_\kappa$ are required to build encryptions of $\pi_1, \ldots, \pi_\kappa$. More precisely,

$$X(\mathcal{O}_i(\boldsymbol{c}, \boldsymbol{c}')) = (x_1 x'_{\sigma_i(1)}, \ldots, x_\kappa x'_{\sigma_i(\kappa)})$$
$$R(\mathcal{O}_i(\boldsymbol{c}, \boldsymbol{c}')) = (r_1 r'_{\sigma_i(1)}, \ldots, r_\kappa r'_{\sigma_i(\kappa)})$$

It then suffices to homomorphically add these encryptions to obtain an encryption of xx'.

‘

Definition 4. *Given $i \in \{1, \ldots, \kappa\}$, OGen(S,i) outputs the expanded representation of the quadratic polynomials $q_1, \ldots, q_{2\kappa}$ defined by*

$$\begin{pmatrix} q_1 \\ \cdots \\ q_{2\kappa} \end{pmatrix} = S^{-1} \begin{pmatrix} \mathcal{L}_{1,2\sigma_i(1)-1} \\ \mathcal{L}_{2,2\sigma_i(1)} \\ \cdots \\ \mathcal{L}_{2\kappa-1,2\sigma_i(\kappa)-1} \\ \mathcal{L}_{2\kappa,2\sigma_i(\kappa)} \end{pmatrix}$$

and $\mathcal{O}_i(\boldsymbol{c}, \boldsymbol{c}') = (q_{i1}(\boldsymbol{c}, \boldsymbol{c}'), \ldots, q_{i,2\kappa}(\boldsymbol{c}, \boldsymbol{c}'))$

As highlighted in Fig. 3, $\mathcal{O}_i(\boldsymbol{c}, \boldsymbol{c}')$ is a valid operator which outputs an encryption of π_i. The multiplicative operator can be then defined as follows

$$\boldsymbol{c} \otimes \boldsymbol{c}' = \mathcal{O}_1(\boldsymbol{c}, \boldsymbol{c}') \oplus \ldots \oplus \mathcal{O}_\kappa(\boldsymbol{c}, \boldsymbol{c}')$$

Proposition 2. *The operator $\otimes$ is a valid multiplicative homomorphic operator.*

Proof. It suffices to show that $\boldsymbol{c}'' \leftarrow \mathcal{O}_i(\boldsymbol{c}, \boldsymbol{c}')$ is an encryption of $\pi_i = \sum_{j=1}^{\kappa} x_i x'_{\sigma_i(j)}$.

$\mathsf{Decrypt}(S, pp, \boldsymbol{c}'')$
$= \sum_{\ell=1}^{\kappa} \frac{r_\ell r'_{\sigma_i(\ell)} x_\ell x_{\sigma_i(\ell)'}}{r_\ell r'_{\sigma_i(\ell)}}$
$= x_1 x'_{\sigma_i(1)} + \ldots + x_\kappa x'_{\sigma_i(\kappa)}$
$= \pi_i$
□

$$\mathcal{O}_i\left(S^{-1}\begin{pmatrix} r_1x_1 \\ r_1 \\ \cdots \\ r_\kappa x_\kappa \\ r_\kappa \end{pmatrix}, S^{-1}\begin{pmatrix} r'_1x'_1 \\ r'_1 \\ \cdots \\ r'_\kappa x'_\kappa \\ r'_\kappa \end{pmatrix}\right) = S^{-1}\begin{pmatrix} r_1 r'_{\sigma_i(1)} x_1 x'_{\sigma_i(1)} \\ r_1 r'_{\sigma_i(1)} \\ \cdots \\ r_\kappa r'_{\sigma_i(\kappa)} x_\kappa x'_{\sigma_i(\kappa)} \\ r_\kappa r'_{\sigma_i(\kappa)} \end{pmatrix}$$

Fig. 3. Description of the operator $\mathcal{O}_i$ showing that $\mathsf{Decrypt}(K, pp, \mathcal{O}_i(\boldsymbol{c}, \boldsymbol{c}')) = \pi_i$

Case $\kappa = 2$. *Given two encryptions $\boldsymbol{c}, \boldsymbol{c}'$ of x, x', we have*

$$\mathcal{O}_1(\boldsymbol{c}, \boldsymbol{c}') = T\begin{pmatrix} r_1r'_1x_1x'_1 \\ r_1r'_1 \\ r_2r'_2x_2x'_2 \\ r_2r'_2 \end{pmatrix} ; \mathcal{O}_2(\boldsymbol{c}, \boldsymbol{c}') = T\begin{pmatrix} r_1r'_2x_1x'_2 \\ r_1r'_2 \\ r_2r'_1x_2x'_1 \\ r_2r'_1 \end{pmatrix}$$

implying that $\boldsymbol{c}'' = \mathsf{Mult}(\boldsymbol{c}, \boldsymbol{c}') \stackrel{def}{=} \mathsf{Add}(\mathcal{O}_1(\boldsymbol{c}, \boldsymbol{c}'), \mathcal{O}_2(\boldsymbol{c}, \boldsymbol{c}'))$ is a valid encryption of xx'. Indeed,

$$\boldsymbol{c}'' = T\begin{pmatrix} r_1^2r'_1r'_2(x_1x'_1 + x_1x'_2) \\ r_1^2r'_1r'_2 \\ r_2^2r'_1r'_2(x_2x'_1 + x_2x'_2) \\ r_2^2r'_1r'_2 \end{pmatrix}$$

and $\mathsf{Decrypt}(K, pp, \boldsymbol{c}'') = x_1x'_1 + x_1x'_2 + x_2x'_1 + x_2x'_2 = (x'_1 + x'_2)(x_1 + x_2) = xx'$.

4.3 Towards a public-key encryption

The classic way (see [Rot11]) to transform a private-key cryptosystem into a public-key cryptosystem consists of publicizing encryptions $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_t$ of known values $x_1, \ldots, x_t$ and using the homomorphic operators to encrypt x. Let $\mathsf{Encrypt1}$ denote this new encryption function. Assuming the IND-CPA security of the private-key cryptosystem, it suffices that $\mathsf{Encrypt1}(pk, x)$ and $\mathsf{Encrypt}(K, pp, x)$ are computationally indistinguishable to ensure the IND-CPA security of the public-key cryptosystem. One can easily check this can be achieved (with overwhelming probability) by choosing $t = \mathcal{O}(\kappa)$.

5 Security analysis

In order to make the basic attack fail,

$$\kappa = \Theta(\lambda)$$

throughout this paper. To simplify our security analysis, we propose minor modifications in the definitions of the encrypting function and the homomorphic operators (see definitions 2, 7 and 8) consisting of replacing S^{-1} by $\det S \times S^{-1}$. It is straightforward to show that our construction remains correct. This is done to ensure that each *value* known by the CPA attacker can be expressed as a polynomial defined over the coefficients of S.

There are classically two sources of randomness *behind* the knowledge of the CPA attacker. The first source of randomness is the internal randomness of KeyGen, i.e. the choice of $K = \{S\}$. The second source of randomness comes from the encryption oracle. After receiving the challenge encryption $\boldsymbol{c}_0 \leftarrow \mathsf{Encrypt}(K, pp, x_0)$, the CPA attacker requests the encryption oracle to get encryptions $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_t$ of arbitrarily chosen plaintexts $x_1, \ldots, x_t \in \mathbb{Z}_n$.

Definition 5. *Let $S \leftarrow \mathsf{KeyGen}(\lambda)$, let $(x_{i1}, r_{i1}, \ldots, x_{i\kappa}, r_{i\kappa})$ be the values (randomly) chosen by the encryption oracle to produce*[4] *$\boldsymbol{c}_i$. For any $\ell \in \{1, \ldots, \kappa\}$, the random vector θ_ℓ is defined by*

$$\theta_\ell = (\boldsymbol{s}_{2\ell-1}, \boldsymbol{s}_{2\ell}, (x_{i\ell}, r_{i\ell})_{i=0,\ldots,t})$$

The random vector $(\theta_1, \ldots, \theta_\kappa)$ is denoted by $\boldsymbol{\theta}$.

The knowledge of the CPA attacker can be represented as a vector $\boldsymbol{\alpha} \in \mathbb{Z}_n^\gamma$, with $\gamma = O(\kappa^4)$ provided $t = \Theta(\kappa)$.

Definition 6. *The CPA attacker's knowledge $(\boldsymbol{c}_0, \ldots, \boldsymbol{c}_t, x_1, \ldots, x_t, \mathsf{Add}, \mathsf{Mult})$ can be represented by a vector $\boldsymbol{\alpha}$, the i^{th} component of $\boldsymbol{\alpha}$ being the evaluation of a polynomial α_i over $\boldsymbol{\theta}$, i.e. $\boldsymbol{\alpha} = (\alpha_1(\boldsymbol{\theta}), \alpha_2(\boldsymbol{\theta}), \ldots) \stackrel{def}{=} \alpha(\boldsymbol{\theta})$.*

The polynomials α_i have intrinsical symmetry properties.

Lemma 2. *Each polynomial α_i is Σ_κ-symmetric (see Definition 1).*

Proof. See [GT20]
□

This result means that $\alpha_i(\theta_1, \ldots, \theta_\kappa) = \alpha_i(\theta_{\sigma(1)}, \ldots, \theta_{\sigma(\kappa)})$ for any $\sigma \in \Sigma_\kappa$. For instance, Add and the operators $\mathcal{O}_1, \ldots, \mathcal{O}_\kappa$ remain unchanged by permuting $\theta_1, \ldots, \theta_\kappa$ according to σ_j, i.e. replacing $(\theta_1, \ldots, \theta_\kappa)$ by $(\theta_j, \ldots, \theta_\kappa, \theta_1, \ldots, \theta_{j-1})$.

[4] $\boldsymbol{c}_i = T(r_{i1}x_{i1}, r_{i1}, \ldots, r_{i\kappa}x_{i\kappa}, r_{i\kappa})$.

5.1 A fundamental result

By mixing Lemma 1 and Lemma 2, we get the following fundamental result.

Proposition 3. *Assume the hardness of factoring, $\pi(\boldsymbol{\theta})$ cannot be evaluated provided π is a polynomial which is not Σ_κ-symmetric. In particular, the CPA attacker cannot recover any:*

1. *coefficient of S,*
2. *product of strictly less than κ coefficients of S,*
3. *polynomial[5] $\mathcal{L}_{i_1} \times \cdots \times \mathcal{L}_{i_t}$ provided $t < \kappa$,*

Proof. A direct consequence of Lemmas 1 and 2.
□

Consider, for instance, the polynomial

$$\phi_D = \sum_{\ell=1}^{\kappa} \mathcal{L}_{2\ell-1} \prod_{\ell' \neq \ell} \mathcal{L}_{2\ell'}$$

can be used to distinguish between encryptions of 0 and encryptions of 1. Clearly, each monomial coefficient of ϕ_D is Σ_κ-symmetric (and thus could be perhaps recovered). However, the expanded representation of ϕ_D (or its multiples) is exponential-size provided $\kappa = \Theta(\lambda)$ and thus cannot be recovered.
By construction, ϕ_D (or its multiples) could nevertheless be efficiently represented with the linear functions $\mathcal{L}_i$ (or O(1)-products of these linear functions). However, these compact semi-factored representations do not deal with symmetric quantities and they cannot be recovered according to Proposition 3. Unfortunately, our scheme suffers from some vulnerabilities detailed in the two following subsections.

5.2 Algebraic attacks based on Grœbner basis

The knowledge of the CPA attacks can be seen as evaluations of polynomials over $\boldsymbol{\theta}$. Hence, our scheme can be seen as the over-defined system of nonlinear equations

$$\begin{aligned} \alpha_1 - \alpha_1(\boldsymbol{\theta}) &= 0 \\ \alpha_2 - \alpha_2(\boldsymbol{\theta}) &= 0 \\ &\cdots \end{aligned}$$

Let I denote the ideal generated by the polynomials $\alpha_i - \alpha_i(\boldsymbol{\theta})$. Computing Grœbner basis [Buc06] of I is relevant to solve such systems of equations.

By using variable elimination technics (e.g. based on Groebner basis), univariate equations dealing with any coefficient s_{ij} of S could be recovered by

[5] and thus cannot be evaluated

computing $I \cap \langle s_{ij} \rangle$. However, thanks to symmetry, such equations are ensured to be nonlinear and thus cannot be solved under the factoring assumption. This is exactly what Proposition 3 encapsulates.

Nevertheless, other attacks based on Grœbner basis can be imagined. This is the object of this section.

A basic example. Similarly to the basic attack of the private-key encryption scheme (see Section 3), the system of equations

$$\phi_D(\boldsymbol{c}_i) - x_i \phi'_D(\boldsymbol{c}_i) = 0 \quad \text{for any } i = 0, \ldots, t \tag{3}$$

can be considered as linear. The number of variables of this linear system being equal $2 \times M(2\kappa, \kappa)$ (see notation of the introduction), it is exponential-size provided $\kappa = \Theta(\lambda)$.

By considering these equations as nonlinear, i.e. defined over the variables s_{ij} and x_0 (assuming $x_1, \ldots, x_t$ are known), attacks based on Grœbner basis could be relevant to recover x_0. We simply consider the ideal I generated by the polynomials $\phi_D(\boldsymbol{c}_i) - x_i \phi'_D(\boldsymbol{c}_i)$, i.e.

$$\sum_{\ell=1}^{\kappa} \langle \boldsymbol{s}_{2\ell-1}, \boldsymbol{c}_i \rangle \prod_{\ell' \neq \ell} \langle \boldsymbol{s}_{2\ell'}, \boldsymbol{c}_i \rangle - x_i \prod_{\ell=1}^{\kappa} \langle \boldsymbol{s}_{2\ell}, \boldsymbol{c}_i \rangle$$

We then eliminate the variables s_{ij} to recover x_0.

We first experiment the case $\kappa = 1$. We measure the running times *w.r.t.* t.

t	4	5	6	7	$\cdots$	10	$\cdots$	15
time(ms)	1.5	1.5	1.5	1.5	$\cdots$	1.6	$\cdots$	1.6

Consider now $\kappa = 2$. We obtain the following running times

t	18	19	20	21	$\cdots$	25	$\cdots$	30
time(s)	780	125	21	21	$\cdots$	22	$\cdots$	21

By noticing that $M(4, 2) = 10$, we notice a major threshold effect when

$$t < 2 \times M(2\kappa, \kappa)$$

In other words, when the system of equations (3) cannot be considered as linear (because the number of equations is too small, i.e. strictly smaller than $2 \times M(2\kappa, \kappa)$), elimination technics seem dramatically not efficient to recover x_0. This would be sufficient to prove the inefficiency of such attacks by recalling that $M(2\kappa, \kappa)$ is exponential in κ. Because of prohibitive running times, this threshold effect was unfortunately not confirmed for higher values of κ, i.e. we did not obtain any result within 24h for $\kappa = 3$. Nevertheless, this clearly suggests the inefficiency of these attacks.

Moreover, the above experiments are intrinsically not efficient. Indeed, the expanded representation of the polynomials ϕ_D and ϕ'_D is exponential *w.r.t.* κ and thus cannot be directly considered. To overcome this, new variables could be introduced. For instance, we introduce the variables $x_{i1}, \ldots, x_{i\kappa}$ for each encryption $\boldsymbol{c}_i$ and replace (3) by the $\kappa + 1$ following equations

$$\langle \boldsymbol{s}_{2\ell-1}, \boldsymbol{c}_i \rangle = x_{i\ell} \cdot \langle \boldsymbol{s}_{2\ell}, \boldsymbol{c}_i \rangle \text{ for any } \ell = 1, \ldots, \kappa$$
$$x_{i1} + \ldots + x_{i\kappa} = x_i$$

This dramatically degrades performance in all our experiments. Indeed, recovering x_0 requires the elimination of the variables x_{ij} which intrinsically leads to the polynomials ϕ_D and ϕ'_D. In our opinion, Grœbner basis are clearly not relevant tools to attack our private-key encryption (without taking into account homomorphic operators) provided

$$\kappa = \Theta(\lambda)$$

Attack 1. We here propose to decrypt the challenge encryption only knowing the homomorphic operators (without any access to the encryption oracle). For concreteness, we consider the ideal I containing the polynomials related to the homomorphic operators and the challenge encryption.
The methods based on Grœbner basis deal with expanded representations. As the number of monomials of each coefficient of S^{-1} (or more precisely $\det S \times S^{-1}$) is exponential *w.r.t.* κ.

A first way to overcome this could consist of introducing new variables $t_{11}, t_{12}, \ldots, t_{2\kappa,2\kappa}$ and replacing S^{-1} by the matrix $T = [t_{ij}]$. This can be done by adding to the ideal I all the equations coming from the equality $T \times S = \mathsf{Id}$. However, this approach dramatically increases the running times in all our experiments.

Let us propose an other approach. For instance, let us consider the operator $(q_1, \ldots, q_{2\kappa}) = \mathcal{O}_1$. By definition,

$$S \begin{pmatrix} q_1 \\ \cdots \\ q_{2\kappa} \end{pmatrix} - \begin{pmatrix} \mathcal{L}_{11} \\ \cdots \\ \mathcal{L}_{2\kappa,2\kappa} \end{pmatrix} = 0$$

The polynomials coming from this equation are only quadratic ensuring polynomial-size expanded representations[6].

By using these basic optimizations (see Fig. 4 for a Sagemath implementation of this attack dealing with the case $\kappa = 2$), the challenge encryption can be decrypted with the following running times:

κ	2	...	6	7	8
$\mathsf{time}(s)$	0.06	...	24	83	330

It is unclear whether these attacks are efficient or not. Nevertheless, they seem more efficient than the basic attack.

[6] The same can be done with the other operators.

Remark 3. The above running times are not improved by including the encryptions $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_t$ in the ideal I. This explains why they were not considered in Attack 1.

Remark 4. Attacks based on Grœbner basis seem totally inefficient if Mult is discarded from I. This enhances our confidence in the security of the additive scheme proposed in [GT20].

5.3 Attacks using specificities of the homomorphic operators

The attacks based on Grœbner basis are general in the sense that they can be mounted whatever the way to define the homomorphic operators is. We here propose a list (hopefully exhaustive) of efficient attacks exploiting specificities of our construction.

Attack 2. By definition of our operator $\otimes$, we can write ($\boldsymbol{u} \sim \boldsymbol{v}$ meaning that $\exists k$ s.t. $\boldsymbol{u} = k\boldsymbol{v}$),

$$\boldsymbol{c} \otimes \boldsymbol{c} \sim S^{-1} \begin{pmatrix} r_1^\kappa x_1(x_1 + x_2 + \ldots + x_\kappa) \\ r_1^\kappa \\ \cdots \\ r_\kappa^\kappa x_\kappa(x_1 + x_2 + \ldots + x_\kappa) \\ r_\kappa^\kappa \end{pmatrix} \sim S^{-1} \begin{pmatrix} r_1^\kappa x_1 x \\ r_1^\kappa \\ \cdots \\ r_\kappa^\kappa x_\kappa x \\ r_\kappa^\kappa \end{pmatrix}$$

This is obviously a disaster in term of security. Indeed, if $\boldsymbol{c}$ is an encryption of 0 then

$$\boldsymbol{c} \otimes \boldsymbol{c} \sim S^{-1} \begin{pmatrix} 0 \\ r_1^\kappa \\ \cdots \\ 0 \\ r_\kappa^\kappa \end{pmatrix}$$

Let $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_\kappa$ be encryptions of 0. To test whether a challenge encryption $\boldsymbol{c}$ is an encryption of 0, it suffices to check that $\boldsymbol{c} \otimes \boldsymbol{c}$ belongs to the vectorial space spanned by the vectors $\boldsymbol{c}_1 \otimes \boldsymbol{c}_1, \ldots, \boldsymbol{c}_\kappa \otimes \boldsymbol{c}_\kappa$.

Attack 3. Let us consider the operator $\mathcal{O}_1$ defined in the previous section, i.e. applying $\mathcal{O}_1$ consists of evaluating the polynomials $q_1, \ldots, q_{2\kappa}$ defined by

$$\begin{pmatrix} q_1 \\ \cdots \\ q_{2\kappa} \end{pmatrix} = S^{-1} \begin{pmatrix} \mathcal{L}_{11} \\ \mathcal{L}_{22} \\ \cdots \\ \mathcal{L}_{2\kappa,2\kappa} \end{pmatrix}$$

As the vector $\boldsymbol{v} = S^{-1}(0, 1, 0, 1, \ldots, 0, 1)$ is the unique vector satisfying $\mathsf{Add}(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{u}$ for any valid encryption $\boldsymbol{u}$, it can be recovered by solving a linear system. It

```
import time
n=97; kappa=2; ka=2*kappa
alpha=[];pi=[];L=[];LL=[];qq=[]
R. <x1,x2,x,s11,s12,s13,s14,s21,s22,s23,s24,s31,s32,s33,s34,s41,s42,s43,s44,u1,u2,u3,u4,v1,v2,v3,v4>
=PolynomialRing(FiniteField(n),27)
u=vector([u1,u2,u3,u4])
v=vector([v1,v2,v3,v4])
S=matrix(4,4,[s11,s12,s13,s14,s21,s22,s23,s24,s31,s32,s33,s34,s41,s42,s43,s44])
SS=matrix(FiniteField(n),4,4,(randint(1, n) for i in range(16)))
TT=SS.inverse()
c=vector([1,2,3,4])
o=S*c
pi=[x1+x2-x]
pi.append(o[0]-x1*o[1])
pi.append(o[2]-x2*o[3])
su=S*u
sv=S*v
ssu=SS*u
ssv=SS*v
L.append(vector([(su[0]*sv[0]),su[1]*sv[1],su[2]*sv[2],su[3]*sv[3] ]))
L.append(vector([(su[0]*sv[1]+su[1]*sv[0]),su[1]*sv[1],su[2]*sv[3]+su[3]*sv[2],su[3]*sv[3] ]))
L.append(vector([(su[0]*sv[2]),su[1]*sv[3],su[2]*sv[0],su[3]*sv[1] ]))
LL.append(vector([(ssu[0]*ssv[0]),ssu[1]*ssv[1],ssu[2]*ssv[2],ssu[3]*ssv[3] ]))
LL.append(vector([(ssu[0]*ssv[1]+ssu[1]*ssv[0]),ssu[1]*ssv[1],ssu[2]*ssv[3]+ssu[3]*ssv[2],ssu[3]*ssv[3] ]))
LL.append(vector([(ssu[0]*ssv[2]),ssu[1]*ssv[3],ssu[2]*ssv[0],ssu[3]*ssv[1] ]))
for i in range(kappa+1):
    q=TT*LL[i]
    for j in range(ka):
        qq.append(q[j])
for t in range(kappa+1):
    for i in range(ka):
        u=[0]*ka
        u[i]=u[i]+1
        for k in range(ka):
            v=[0]*ka
            v[k]=v[k]+1
            Z=[0]*ka
            for d in range(ka):
                Z[d]=qq[ka*t+d].coefficient(u1:u[0],u2:u[1],u3:u[2],u4:u[3],v1:v[0],v2:v[1],v3:v[2],v4:v[3])
            ZZ=vector(Z)
            Y=S*ZZ
            for m in range(ka):
                p=Y[m]-L[t][m].coefficient(u1:u[0],u2:u[1],u3:u[2],u4:u[3],v1:v[0],v2:v[1],v3:v[2],v4:v[3])
                alpha.append(p)
tp1=time.clock()
IF=R.ideal(alpha+pi)
H=IF.elimination_ideal([x1,x2,s12,s13,s14,s21,s22,s23,s24,s31,s32,s33,s34,s41,s42,s43,s44])
print(H)
tp2=time.clock()
print(tp2-tp1)
```

Fig. 4. Implementation of Attack 1 : a *SageMath* program decrypting the challenge encryption $\boldsymbol{c} = (1, 2, 3, 4)$ only knowing the homomorphic operators in the case $\kappa = 2$. We obtained "Ideal (s11*x - 31*s11, s11^3 + 6*s11^2 - 25*s11) of Multivariate Polynomial Ring in x1, x2, x, s11, s12, s13, s14, s21, s22, s23, s24, s31, s32, s33, s34, s41, s42, s43, s44, u1, u2, u3, u4, v1, v2, v3, v4 over Finite Field of size 97".

follows that $\mathcal{O}_1(\boldsymbol{c}, \boldsymbol{v})$ is equal to the vector $\boldsymbol{w} = S^{-1}(0, r_1, 0, r_2, \ldots, 0, r_\kappa)$. By solving the equation $\mathsf{Add}(\boldsymbol{u}, \boldsymbol{w}) = \boldsymbol{c}$, we get the vector

$$\widetilde{\boldsymbol{c}} = S^{-1} \begin{pmatrix} x_1 \\ 1 \\ \cdots \\ x_\kappa \\ 1 \end{pmatrix}$$

satisfying $\langle \boldsymbol{s}_1 + \boldsymbol{s}_3 + \cdots + \boldsymbol{s}_{2\kappa-1}, \widetilde{\boldsymbol{c}} \rangle = x$. This leads to an efficient attack consisting of solving a linear system of size 2κ.

Attack 4. Let us consider the vector $\boldsymbol{v} = \boldsymbol{u}_{\kappa-1}$ built as follows:

- $\boldsymbol{u}_0 = \boldsymbol{c}$
- $\boldsymbol{u}_i = \mathcal{O}_{i+1}(\boldsymbol{u}_{i-1}, \boldsymbol{c})$ for any $i = 1, \ldots, \kappa - 1$

By construction,

$$\boldsymbol{v} = S^{-1} \begin{pmatrix} r_1 \ldots r_{\kappa-1} x_1 \ldots x_{\kappa-1} \\ r_1 \ldots r_{\kappa-1} \\ r_2 \ldots r_\kappa x_2 \ldots x_\kappa \\ r_2 \ldots r_\kappa \\ \cdots \\ r_\kappa r_1 \ldots r_{\kappa-2} x_\kappa x_1 \ldots x_{\kappa-2} \\ r_\kappa r_1 \ldots r_{\kappa-2} \end{pmatrix}$$

Il follows that

$$\begin{aligned} \varphi(\boldsymbol{c}, \boldsymbol{v}) &\stackrel{\mathsf{def}}{=} \mathcal{L}_{14}(\boldsymbol{c}, \boldsymbol{v}) + \mathcal{L}_{36}(\boldsymbol{c}, \boldsymbol{v}) + \ldots + \mathcal{L}_{2\kappa-1,2}(\boldsymbol{c}, \boldsymbol{v}) \\ &= r_1 \cdots r_\kappa \, (x_1 + \ldots + x_\kappa) \\ &= \phi_D(\boldsymbol{c}) \end{aligned}$$

can be used to distinguish encryptions of 0 from random ones. Moreover, as this polynomial is quadratic, it expanded representation can be polynomially recovered by solving a linear system (by considering sufficiently many encryptions of 0).

Attack 5. Thanks to the operator $\mathcal{O}_1$, one can obtain vectors $\boldsymbol{c}_1 = \boldsymbol{c}, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_{4\kappa}$ satisfying

$$\boldsymbol{c}_i = S^{-1} \begin{pmatrix} r_1^i x_1^i \\ r_1^i \\ \cdots \\ r_\kappa^i x_\kappa^i \\ r_\kappa^i \end{pmatrix}$$

As $\langle \boldsymbol{s}_1, \boldsymbol{c}_i \rangle - x_1^i \langle \boldsymbol{s}_2, \boldsymbol{c}_i \rangle = 0$, the determinant of the following matrix

$$\begin{pmatrix} c_{11} & \dots & c_{1,2\kappa} & -x_1c_{11} & \dots & -x_1c_{1,2\kappa} \\ c_{21} & \dots & c_{2,2\kappa} & -x_1^2c_{21} & \dots & -x_1^2c_{2,2\kappa} \\ \dots & & & & & \\ c_{4\kappa,1} & \dots & c_{4\kappa,2\kappa} & -x_1^{4\kappa}c_{4\kappa,1} & \dots & -x_1^{4\kappa}c_{4\kappa,2\kappa} \end{pmatrix}$$

is null. This leads to an univariate degree-$\kappa(6\kappa+1)$ equation in x_1. Thanks to symmetry properties, $x_2, \dots, x_\kappa$ are also solutions of this equation. In other words, the equation is

$$(x - x_1)^{6\kappa+1} \cdots (x - x_\kappa)^{6\kappa+1} = 0$$

with overwhetming probability. Thus, $x = x_1 + \ldots + x_\kappa$ can be recovered in polynomial-time (it can be recovered from the coefficient of x^{d-1}, d being the degree of the equation).

5.4 An assumption

The attacks 2,3,4,5 belong to the class of attacks by linearization defined in [GT20]. Informally, these attacks consist of using the homomorphic operators (in an arbitrary way) to build new encryptions $v_1, \dots, v_r$ from the challenge encryption $\boldsymbol{c}$ (and known encryptions). This leads to an efficient attack if there exists a *small* polynomial ϕ such that $\phi(v_1, \dots, v_r) = 0$ if and only if $\boldsymbol{c}$ is an encryption of 0. Indeed, the monomial coefficients of ϕ can be efficiently recovered by solving a linear system (as done in the basic attack). It can be then used to distinguish encryptions of 0 from random ones.

Let us try to find a common point between all the attacks of Section 5.3. Assume $\boldsymbol{c}'' = \mathcal{O}(\boldsymbol{c}, \boldsymbol{c}')$ where $\mathcal{O}$ is one of the (naive) operators previously defined. By construction,

$$X(\boldsymbol{c}'') = \phi^{\mathcal{O},X}(X(\boldsymbol{c}), X(\boldsymbol{c}')) = \left(\phi_i^{\mathcal{O},X}(X(\boldsymbol{c}), X(\boldsymbol{c}'))\right)_{i=1,\dots,\kappa}$$
$$R(\boldsymbol{c}'') = \phi^{\mathcal{O},R}(R(\boldsymbol{c}), R(\boldsymbol{c}')) = \left(\phi_i^{\mathcal{O},R}(R(\boldsymbol{c}), R(\boldsymbol{c}'))\right)_{i=1,\dots,\kappa}$$

where $\phi_i^{\mathcal{O},X}$ and $\phi_i^{\mathcal{O},R}$ are public polynomials. For instance,

$$\phi^{\mathsf{Add},X}(X(\boldsymbol{c}), X(\boldsymbol{c}')) = X(\boldsymbol{c}) + X(\boldsymbol{c}')$$

All the attacks of Section 5.3 exploit the fact the functions $\phi_i^{\mathcal{O},X}$ and $\phi_i^{\mathcal{O},R}$ are known and deterministic. By assuming that these conditions are necessary to mount efficient attacks, it would be relevant to build operators where $\phi_i^{\mathcal{O},X}$ and $\phi_i^{\mathcal{O},R}$ are probabilistic functions. This leads us to formulate the following assumption.

Assumption 1 ***(Informal).*** *Our construction would be IND-CPA secure if the functions $\phi_i^{\mathcal{O},X}$ and $\phi_i^{\mathcal{O},R}$ were probabilistic.*

This is obviously not possible to build probabilistic homomorphic operators. However, the new homomorphic operators developed in the next section will ensure that $X(\boldsymbol{c}'')$ does not only depend on $X(\boldsymbol{c})$ and $X(\boldsymbol{c}')$ but it also depends on $R(\boldsymbol{c}'), R(\boldsymbol{c}')$. Similarly $R(\boldsymbol{c}'')$ will not only depend on $R(\boldsymbol{c})$ and $R(\boldsymbol{c})$ but it will also depend on $X(\boldsymbol{c}'), X(\boldsymbol{c}')$. In other words,

$$X(\boldsymbol{c}'') = \left(\phi_i^{\mathcal{O},X}\left(R(\boldsymbol{c}), R(\boldsymbol{c}'), X(\boldsymbol{c}), X(\boldsymbol{c}')\right)\right)_{i=1,\ldots,\kappa}$$
$$R(\boldsymbol{c}'') = \left(\phi_i^{\mathcal{O},R}\left(R(\boldsymbol{c}), R(\boldsymbol{c}'), X(\boldsymbol{c}), X(\boldsymbol{c}')\right)\right)_{i=1,\ldots,\kappa}$$

Roughly speaking, the two independent sources of randomness associated to the functions X and R will be mixed in the construction of $\boldsymbol{c}''$. The functions $\phi_1^{\mathcal{O},X}, \ldots, \phi_\kappa^{\mathcal{O},X}, \phi_1^{\mathcal{O},R}, \ldots, \phi_\kappa^{\mathcal{O},R}$ will be randomly and independently chosen in a given set of rational functions. Moreover, Proposition 3 can be straightforwardly extended to show that these functions cannot be recovered under the factoring assumption.

6 New homomorphic operators

Notation. *$\mathcal{P}$ and $\mathcal{Q}$ will refer to subsets of homogeneous polynomials belonging to $\mathbb{Z}_n[U_1, \ldots, U_{2\kappa}, V_1, \ldots, V_{2\kappa}]$. The choice of these sets will be discussed later. In the following of the paper, the operators $\mathsf{Add}, \mathcal{O}_1, \ldots, \mathcal{O}_\kappa$ developed in the previous sections will be denoted by $\mathsf{Add}^{naive}, \mathcal{O}_1^{naive}, \ldots, \mathcal{O}_\kappa^{naive}$.*

6.1 The additive operator

Ideally, we would like to build an encryption $\boldsymbol{c}''=\mathsf{Add}(\boldsymbol{c}, \boldsymbol{c}')$ defined by

$$\boldsymbol{c}'' = S^{-1}\begin{pmatrix} \eta_1 r_1 r_1'(x_1 + x_1' + \nu_1) \\ \eta_1 r_1 r_1' \\ \cdots \\ \eta_\kappa r_\kappa r_\kappa'(x_\kappa + x_\kappa' + \nu_\kappa) \\ \eta_\kappa r_\kappa r_\kappa' \end{pmatrix}$$

where η_ℓ, ν_ℓ are randomly chosen in $\mathbb{Z}_n$ ensuring that $\nu_1 + \ldots + \nu_\kappa = 0$. This would be sufficient to ensure that $\boldsymbol{c}''$ is drawn according to a probability distribution indistinguishable from $\mathsf{Encrypt}(K, pp, x + x')$.

However, it is obviously not possible to build such encryptions with deterministic operators. We propose to define η_ℓ and ν_ℓ as evaluations of secret polynomials over $\boldsymbol{c}, \boldsymbol{c}'$.

Definition 7. *$\mathsf{AddGen}(S)$ randomly chooses polynomials $\eta_1, \ldots, \eta_\kappa \in \mathcal{P}$ and $\nu_0, \ldots, \nu_\kappa \in \mathcal{Q}$ s.t. $\nu_1 + \ldots + \nu_\kappa = 0$ and outputs the expanded representation of the polynomials $q_1, \ldots, q_{2\kappa}$ defined by*

$$\begin{pmatrix} q_1 \\ \cdots \\ q_{2\kappa} \end{pmatrix} = S^{-1} \begin{pmatrix} \eta_1 \left(\nu_0(\mathcal{L}_{12} + \mathcal{L}_{21}) + \nu_1 \mathcal{L}_{22}\right) \\ \eta_1 \nu_0 \mathcal{L}_{22} \\ \cdots \\ \eta_\kappa \left(\nu_0(\mathcal{L}_{2\kappa-1,2\kappa} + \mathcal{L}_{2\kappa,2\kappa-1}) + \nu_\kappa \mathcal{L}_{2\kappa,2\kappa}\right) \\ \eta_\kappa \nu_0 \mathcal{L}_{2\kappa,2\kappa} \end{pmatrix}$$

The operator $\mathsf{Add} \leftarrow \mathsf{AddGen}(S)$ consists of evaluating the polynomials $q_1, \ldots, q_{2\kappa}$, i.e.

$$\mathsf{Add}(\boldsymbol{u}, \boldsymbol{v}) = (q_1(\boldsymbol{u}, \boldsymbol{v}), \ldots, q_{2\kappa}(\boldsymbol{u}, \boldsymbol{v}))$$

It follows that the running time of Add is polynomial as long as the expanded representation of the polynomials η_ℓ, ν_ℓ is polynomial-size. It is typically the case provided their degree is $O(1)$.

Proposition 4. *The operator $\oplus$ is a valid additive homomorphic operator.*

Proof. Let $\boldsymbol{c}''$=$\mathsf{Add}(\boldsymbol{c}, \boldsymbol{c}')$.

$$\boldsymbol{c}'' = S^{-1} \begin{pmatrix} \eta_1(\boldsymbol{c}, \boldsymbol{c}') r_1 r_1' \left(\nu_0(\boldsymbol{c}, \boldsymbol{c}')(x_1 + x_1') + \nu_1(\boldsymbol{c}, \boldsymbol{c}')\right) \\ \eta_1(\boldsymbol{c}, \boldsymbol{c}') \nu_0(\boldsymbol{c}, \boldsymbol{c}') r_1 r_1' \\ \cdots \\ \eta_\kappa(\boldsymbol{c}, \boldsymbol{c}') r_\kappa r_\kappa' (\nu_0(\boldsymbol{c}, \boldsymbol{c}')(x_\kappa + x_\kappa') + \nu_\kappa(\boldsymbol{c}, \boldsymbol{c}')) \\ \eta_\kappa(\boldsymbol{c}, \boldsymbol{c}') \nu_0(\boldsymbol{c}, \boldsymbol{c}') r_\kappa r_\kappa' \end{pmatrix}$$

According to Theorem 1, $(\eta_\ell \nu_0)(\boldsymbol{c}, \boldsymbol{c}') = 0$ holds with negligible probability. Thus,

$$\begin{aligned} & \mathsf{Decrypt}(S, pp, \boldsymbol{c}'') \\ =& \sum_{\ell=1}^{\kappa} \frac{\eta_\ell(\boldsymbol{c}, \boldsymbol{c}') r_\ell r_\ell' \left(\nu_0(\boldsymbol{c}, \boldsymbol{c}')(x_\ell + x_\ell') + \nu_\ell(\boldsymbol{c}, \boldsymbol{c}')\right)}{\eta_\ell(\boldsymbol{c}, \boldsymbol{c}') \nu_0(\boldsymbol{c}, \boldsymbol{c}') r_\ell r_\ell'} \\ =& x_1 + x_1' + \frac{\nu_1(\boldsymbol{c}, \boldsymbol{c}')}{\nu_0(\boldsymbol{c}, \boldsymbol{c}')} + \ldots + x_\kappa + x_\kappa' + \frac{\nu_\kappa(\boldsymbol{c}, \boldsymbol{c}')}{\nu_0(\boldsymbol{c}, \boldsymbol{c}')} \\ =& x_1 + x_1' + \ldots + x_\kappa + x_\kappa' + \frac{\nu_1(\boldsymbol{c}, \boldsymbol{c}') + \ldots + \nu_\kappa(\boldsymbol{c}, \boldsymbol{c}')}{\nu_0(\boldsymbol{c}, \boldsymbol{c}')} \\ =& x_1 + x_1' + \ldots + x_\kappa + x_\kappa' \\ =& x + x' \end{aligned}$$

□

Remark 5. The role of ν_0 is to ensure that the polynomials $q_1, \ldots, q_{2\kappa}$ are homogeneous.

Remark 6. The naive operator Add^{naive} is obtained by setting $\nu_0 = \eta_1 = \cdots = \eta_\kappa = 1$, $\nu_1 = \cdots = \nu_\kappa = 0$.

6.2 A multiplicative operator

Similarly to the operator Add^{naive}, the naive operators $\mathcal{O}_i^{naive}$ are modified as follows:

Definition 8. *Given $i \in \{1, \ldots, \kappa\}$, $\mathsf{OGen}(S, i)$ randomly chooses polynomials $\eta_1, \ldots, \eta_\kappa \in \mathcal{P}$ and $\nu_0, \ldots, \nu_\kappa \in \mathcal{Q}$ and outputs the expanded representation of the polynomials $q_1, \ldots, q_{2\kappa}$ defined by*

$$\begin{pmatrix} q_1 \\ \cdots \\ q_{2\kappa} \end{pmatrix} = S^{-1} \begin{pmatrix} \eta_1 \left(\nu_0 \mathcal{L}_{1,2\sigma_i(1)-1} + \nu_1 \mathcal{L}_{2,2\sigma_i(1)}\right) \\ \eta_1 \nu_0 \mathcal{L}_{2,2\sigma_i(1)} \\ \cdots \\ \eta_\kappa \left(\nu_0 \mathcal{L}_{2\kappa-1,2\sigma_i(\kappa)-1} + \nu_\kappa \mathcal{L}_{2\kappa,2\sigma_i(\kappa)}\right) \\ \eta_\kappa \nu_0 \mathcal{L}_{2\kappa,2\sigma_i(\kappa)} \end{pmatrix}$$

The operator $\mathcal{O}_i \leftarrow \mathsf{OGen}(S, i)$ consists of evaluating the polynomials $q_1, \ldots, q_{2\kappa}$, i.e. $\mathcal{O}_i(\boldsymbol{u}, \boldsymbol{v}) = (q_1(\boldsymbol{u}, \boldsymbol{v}), \ldots, q_{2\kappa}(\boldsymbol{u}, \boldsymbol{v}))$. The multiplicative operator can be then defined as follows

$$\boldsymbol{c} \otimes \boldsymbol{c}' = \mathcal{O}_1(\boldsymbol{c}, \boldsymbol{c}') \oplus \ldots \oplus \mathcal{O}_\kappa(\boldsymbol{c}, \boldsymbol{c}')$$

Proposition 5. *The operator $\otimes$ is a valid multiplicative homomorphic operator.*

Proof. It suffices to show that $\boldsymbol{c}'' \leftarrow \mathcal{O}_i(\boldsymbol{c}, \boldsymbol{c}')$ is an encryption of $\pi_i = \sum_{j=1}^{\kappa} x_i x'_{\sigma_i(j)}$. According to Theorem 1, $(\eta_\ell \nu_0)(\boldsymbol{c}, \boldsymbol{c}') = 0$ holds with negligible probability. Thus,

$$\begin{aligned}
&\mathsf{Decrypt}(S, pp, \boldsymbol{c}'') \\
=&\sum_{\ell=1}^{\kappa} \frac{\eta_\ell(\boldsymbol{c}, \boldsymbol{c}') r_\ell r'_{\sigma_i(\ell)} \left(\nu_0(\boldsymbol{c}, \boldsymbol{c}') x_\ell x_{\sigma_i(\ell)'} + \nu_\ell(\boldsymbol{c}, \boldsymbol{c}')\right)}{\eta_\ell(\boldsymbol{c}, \boldsymbol{c}') \nu_0(\boldsymbol{c}, \boldsymbol{c}') r_\ell r'_{\sigma_i(\ell)}} \\
=&x_1 x'_{\sigma_i(1)} + \frac{\nu_1(\boldsymbol{c}, \boldsymbol{c}')}{\nu_0(\boldsymbol{c}, \boldsymbol{c}')} + \ldots + x_\kappa x'_{\sigma_i(\kappa)} + \frac{\nu_\kappa(\boldsymbol{c}, \boldsymbol{c}')}{\nu_0(\boldsymbol{c}, \boldsymbol{c}')} \\
=&x_1 x'_{\sigma_i(1)} + \ldots + x_\kappa x'_{\sigma_i(\kappa)} + \frac{\nu_1(\boldsymbol{c}, \boldsymbol{c}') + \ldots + \nu_\kappa(\boldsymbol{c}, \boldsymbol{c}')}{\nu_0(\boldsymbol{c}, \boldsymbol{c}')} \\
=&x_1 x'_{\sigma_i(1)} + \ldots + x_\kappa x'_{\sigma_i(\kappa)} \\
=&\pi_i
\end{aligned}$$

□

6.3 Increasing randomness for almost free

In this section, we propose to *increase* randomness in our homomorphic operators without degrading performance. We will experimentally show that this added randomness strongly increases the running time of Grœbner basis attacks (see Section 7.2).

Operator Rand. We can easily modify the operators $\mathcal{O}_i$ (the same can be done for Add) in order to output encryptions valid under a randomly chosen key T_i (instead of S). For instance, the operator $\mathcal{O}_1$ can be redefined as follows

$$\mathcal{O}_1^{S\to T_1} = T_1^{-1}\begin{pmatrix}\eta_1\left(\nu_0\mathcal{L}_{11}+\nu_1\mathcal{L}_{22}\right)\\ \eta_1\nu_0\mathcal{L}_{22}\\ \cdots\\ \eta_\kappa\left(\nu_0\mathcal{L}_{2\kappa-1,2\kappa-1}+\nu_\kappa\mathcal{L}_{2\kappa,2\kappa}\right)\\ \eta_\kappa\nu_0\mathcal{L}_{2\kappa,2\kappa}\end{pmatrix}$$

By construction, $\boldsymbol{c}'' = \mathcal{O}_i^{S\to T_i}(\boldsymbol{c},\boldsymbol{c}')$ is a valid encryption of π_i under the key T_i. We then develop operators $\mathsf{Rand}^{T_i\to T_i'}$ to switch keys and to *randomize* encryptions (without modifying the plaintexts). Let $\mathcal{L}_j^i$ denote the linear function $\mathcal{L}_j^i(\boldsymbol{u}) = \langle \boldsymbol{t}_{ij}, \boldsymbol{u}\rangle$ where $\boldsymbol{t}_{ij}$ is the j^{th} row of T_i. The operator $\mathsf{Rand}^{T_i\to T_i'}$ can be defined as follows:

$$\mathsf{Rand}^{T_i\to T_i'} = T_i'^{-1}\begin{pmatrix}\eta_1\left(\nu_0\mathcal{L}_1^i+\nu_1\mathcal{L}_2^i\right)\\ \eta_1\nu_0\mathcal{L}_2^i\\ \cdots\\ \eta_\kappa\left(\nu_0\mathcal{L}_{2\kappa-1}^i+\nu_\kappa\mathcal{L}_{2\kappa}^i\right)\\ \eta_\kappa\nu_0\mathcal{L}_{2\kappa}^i\end{pmatrix}$$

where η_ℓ and ν_ℓ are randomly chosen variate-2κ polynomials ensuring that $\nu_1+\ldots+\nu_\kappa = 0$. By construction $\boldsymbol{c}''' = \mathsf{Rand}^{T_i\to T_i'}(\boldsymbol{c}'')$ is a valid encryption of π_i under the key T_i'. This randomization can be chained, i.e. by considering operators $\mathsf{Rand}^{T_i\to T_i'}$, $\mathsf{Rand}^{T_i'\to T_i''}$, $\ldots$, $\mathsf{Rand}^{T_i^{[\gamma]}\to S}$. We can thus define the function $\mathsf{OGen}(S,i,\gamma)$, extending the function $\mathsf{OGen}(S,i)$, which outputs the operator $\mathcal{O}_i^{[\gamma]}$ defined by

$$\mathcal{O}_i^{[\gamma]}(\boldsymbol{c},\boldsymbol{c}') = \mathsf{Rand}^{T_i^{[\gamma]}\to S}\left(\ldots\left(\mathsf{Rand}^{T_i'\to T_i''}\left(\mathsf{Rand}^{T_i\to T_i'}\left(\mathcal{O}_i^{S\to T_i}(\boldsymbol{c},\boldsymbol{c}')\right)\right)\right)\ldots\right)$$

It should be noticed that considering operators Rand does not affect so much the running time of the homomorphic operators: they run at least κ faster than the operators $\mathcal{O}_i$. For instance, if the polynomials η_ℓ, ν_ℓ are linear then $\mathcal{O}_i^{[\gamma]}$ runs in

$$O(\kappa^5+\gamma\kappa^4)$$

Remark 7. The ideas of this section could be used to control homomorphic computations. Consider a set of encryptions $\boldsymbol{c}_1,\ldots,\boldsymbol{c}_t$ valid under randomly chosen keys $S_1,\ldots,S_t$. Given an arithmetic circuit ϕ and a randomly chosen key T, relevant operators can be developed to compute $\boldsymbol{c} = \mathsf{Eval}(\phi,\boldsymbol{c}_1,\ldots,\boldsymbol{c}_t)$ valid under T but *nothing else* (under T).

7 Security analysis

In our security analysis and in all our experiments, the polynomials ν_ℓ, η_ℓ will be linear. More precisely, ν_ℓ, η_ℓ will be randomly chosen in the sets of linear polynomials respectively $\mathcal{P}, \mathcal{Q} \subset \mathbb{Z}_n[U_1,\ldots,U_{2\kappa},V_1,\ldots,V_{2\kappa}]$ defined by:

– $\mathcal{P} = \{a_1U_1 + \ldots + a_{2\kappa}U_{2\kappa} | a_i \in \mathbb{Z}_n\}$
– $\mathcal{Q} = \{a_1V_1 + \ldots + a_{2\kappa}V_{2\kappa} | a_i \in \mathbb{Z}_n\}$

Under this choice, each polynomial associated to each homomorphic operator is in the form

$$\sum_{(i,j,k,\ell)\in\{1,\ldots,2\kappa\}^4 : i\leq j, k\leq \ell} a_{ijk\ell} U_i U_j V_k V_\ell$$

7.1 Security vs Assumption 1

In the construction of our new homomorphic operators, symmetry properties are kept. Hence, Proposition 3 can be straightforwardly extended to our new settings. To achieve this, it suffices to incorporate the internal randomness of AddGen and OGen in $\boldsymbol{\theta}$.

Proposition 6. *Assuming the hardness of factoring, $\pi(\boldsymbol{\theta})$ cannot be evaluated provided π is a polynomial which is not Σ_κ-symmetric. In particular, the CPA attacker cannot recover any:*

1. *coefficient of S,*
2. *product of strictly less than κ coefficients of S,*
3. *polynomial[7] $\mathcal{L}_{i_1} \times \cdots \times \mathcal{L}_{i_t}$ provided $t < \kappa$,*
4. *polynomial ν_ℓ, η_ℓ for any $\ell = 1, \ldots, \kappa$*

Remark 8. This proposition does not ensure anything about ν_0.

Let $\boldsymbol{c}'' = \mathcal{O}(\boldsymbol{c}, \boldsymbol{c}')$ with $\mathcal{O} \in \{\mathcal{O}_1, \ldots, \mathcal{O}_\kappa, \mathsf{Add}\}$. For instance, if $\mathcal{O} = \mathsf{Add}$ then we have

$$X(\boldsymbol{c}'') = \left(x_1 + x_1' + \frac{\nu_1(\boldsymbol{c}, \boldsymbol{c}')}{\nu_0(\boldsymbol{c}, \boldsymbol{c}')}, \ldots, x_\kappa + x_\kappa' + \frac{\nu_\kappa(\boldsymbol{c}, \boldsymbol{c}')}{\nu_0(\boldsymbol{c}, \boldsymbol{c}')}\right)$$
$$R(\boldsymbol{c}'') = \nu_0(\boldsymbol{c}, \boldsymbol{c}')\,(r_1 r_1' \eta_1(\boldsymbol{c}, \boldsymbol{c}'), \ldots, r_\kappa r_\kappa' \eta_\kappa(\boldsymbol{c}, \boldsymbol{c}'))$$

Hence, $X(\boldsymbol{c}'')$ does not only depend on $X(\boldsymbol{c}), X(\boldsymbol{c}')$ and $R(\boldsymbol{c}'')$ does not only depend on $R(\boldsymbol{c}), R(\boldsymbol{c}')$, i.e.

$$X(\boldsymbol{c}'') = \phi^{\mathcal{O},X}(X(\boldsymbol{c}), X(\boldsymbol{c}'), R(\boldsymbol{c}), R(\boldsymbol{c}'))$$
$$R(\boldsymbol{c}'') = \phi^{\mathcal{O},R}(X(\boldsymbol{c}), X(\boldsymbol{c}'), R(\boldsymbol{c}), R(\boldsymbol{c}'))$$

It follows that $X(\boldsymbol{c}'')$ is a probabilistic function (randomness coming from $R(\boldsymbol{c})$, $R(\boldsymbol{c}')$) of $X(\boldsymbol{c}), X(\boldsymbol{c}')$ and $R(\boldsymbol{c}'')$ is a probabilistic function of $R(\boldsymbol{c})$, $R(\boldsymbol{c}')$. The same remark can be done for the other operators. Moreover, Proposition 3 ensures that these functions cannot be recovered under the factoring assumption (the polynomials ν_ℓ and η_ℓ cannot be recovered). According to Assumption 1, this suggests IND-CPA security.

[7] and thus cannot be evaluated

7.2 Algebraic attacks based on Grœbner basis

We here propose new experiments clearly suggesting that algebraic attacks based on variable elimination technics are not efficient. First of all, Proposition 6 can be invoked to exclude a large class of attacks. For instance, coefficients of S cannot be recovered by using such attacks. Indeed, even if univariate equations dealing with coefficients of S can be recovered, symmetry ensures that these equations are not linear and the factoring assumption ensures they cannot be solved. This is fundamental in our security analysis.

Proposition 6 *does not say* anything about the polynomial ν_0. In consequence, this polynomial will be assumed to be known. In all our experiments, we will set

$$\nu_0(\boldsymbol{u}, \boldsymbol{v}) = u_1 + \ldots + u_{2\kappa}$$

As done in Section 5.2, we consider the ideal I generated by the polynomials derived from the homomorphic operators and the ones derived from the challenge encryption $\boldsymbol{c}$, i.e.

$$\begin{array}{l} \langle \boldsymbol{s}_{2\ell-1}, \boldsymbol{c} \rangle = x_\ell \cdot \langle \boldsymbol{s}_{2\ell}, \boldsymbol{c} \rangle \quad \text{for any } \ell = 1, \ldots, \kappa \\ x_1 + \ldots + x_\kappa = x \end{array}$$

The *basic attack* exploits the *basic equation* $\phi_D(\boldsymbol{c}) - x\phi'_D(\boldsymbol{c}) = 0$, i.e.

$$\sum_{\ell=1}^{\kappa} \langle \boldsymbol{s}_{2\ell-1}, \boldsymbol{c} \rangle \prod_{\ell' \neq \ell} \langle \boldsymbol{s}_{2\ell'}, \boldsymbol{c} \rangle - x \prod_{\ell=1}^{\kappa} \langle \boldsymbol{s}_{2\ell}, \boldsymbol{c} \rangle = 0$$

By construction, this equation is exponential-size provided $\kappa = \Theta(\lambda)$. However, *shorter* equations could be efficiently recovered with the introduction of the homomorphic operators. We are here interested in recovering any equation in the form

$$\phi(\boldsymbol{c}, x, S, \mathsf{coeff}) = 0$$

where ϕ is a *short* polynomial[8] and coeff represents the monomial coefficients of all the polynomials ν_ℓ, η_ℓ involved in homomorphic operators. Similarly to the basic attack, such equations lead to attacks consisting of solving linear systems. More precisely, by considering encryptions $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_t$ of known values $y_1, \ldots, y_t$, we obtain the following system of equations

$$\begin{cases} \phi(\boldsymbol{c}, x, S, \mathsf{coeff}) = 0 \\ \\ \phi(\boldsymbol{c}_i, y_i, S, \mathsf{coeff}) = 0 \quad \text{for any } i = 1, \ldots, t \end{cases}$$

By considering this system as linear, univariate equations only dealing with x can be recovered provided t is sufficiently large. Such equations would be relevant to distinguish between $x = 0$ and $x = 1$.

[8] $\phi(\boldsymbol{c}, x, S, \mathsf{coeff}) = \sum_i \phi_i(\boldsymbol{c}, S, \mathsf{coeff}) \times x^i$ with ϕ_i not null for at least one $i \geq 1$.

Such attacks are efficient provided the expanded representation of ϕ is polynomial-size. Ideally, an equation in the form

$$x - a = 0$$

or more generally $\psi(\boldsymbol{c}, x, S, \mathsf{coeff}) \times (x - a) = 0$ would allow the attacker to immediately conclude that $x = a$ without requesting the oracle encryption, i.e. without considering the encryptions $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_t$. As expected, we obtained such equations.

To get such equations, it suffices to eliminate the variables $x_1, \ldots, x_\kappa$. A *sageMath* program dealing with the case $\kappa = 2$ can be found in Appendix C. While this takes only 0.06 seconds with the naive operators in the case $\kappa = 2$, 30s were required with the new ones and approximatively $6h$ for $\kappa = 3$. This is encouraging but not sufficient to conclude this attack is not polynomial.

We carried on other experiments in this sense. We considered the operators $\mathcal{O}_i^{[\gamma]}$ introduced in Section 6.3, choosing $\gamma = 1$. This dramatically affects the running time, e.g. 20 hours in the case $\kappa = 2$. It is tempting to assume that the running time of these attacks is exponential *w.r.t.* γ. To enhance this idea, we propose other experiments dealing with higher values of γ.

Experiments highlighting the role of the parameter γ. Throughout this experiment, we set $\kappa = 2$. In order to not have prohibitive running times and thus to consider higher values of γ, we will consider the naive operators Add^{naive}, $\mathcal{O}_1^{naive}$ and $\mathcal{O}_2^{naive}$ (instead of the new versions of these operators) randomized as explained in Section 6.3: in particular the operators $\mathsf{Rand}^{T \to T'}$ are defined exactly in the same manner[9]. We obtain the following running times.

γ	0	1	2	3
$\mathsf{time}(s)$	0.06	423	8200	179000

These running times clearly confirm our intuition. Nevertheless, improvements could be perhaps obtained by reducing the number of variables in the ideal I. For instance, some/all operators Rand (the polynomials derived from these operators) can be simply removed from I (more exactly, not used to generate I). We propose *to break* the chains of operators Rand by removing at least one operator Rand in each chain (associated to each operator Add^{naive}, $\mathcal{O}_1^{naive}$ and $\mathcal{O}_2^{naive}$). We observe that the running times are significantly degraded. More interesting, there does not apparently exist any equation shorter than the basic one. In particular, x cannot be recovered anymore. Roughly speaking, this means that information is lost when *randomizing chains are broken*. This suggests that the running times previously obtained cannot be improved by considering sub-ideals $I' \subset I$.

[9] For instance, $\mathsf{Add}^{naive, S \to T_0}$ returns encryptions valid under a randomly chosen key T_0. These encryptions are then randomized by applying a *chain* of operators $\mathsf{Rand}^{T_i \to T_{i+1}}$ with $T_\gamma = S$.

Claim 1. *Algebraic attacks based on Grœbner basis are inefficient provided*[10] $\kappa = \Theta(\lambda)$ *and* $\gamma = \Theta(\lambda)$.

Discussion. This claim is only based on experimental results. Further investigations should be obviously done in this sense. Maybe subs-systems of equations could be considered in order to reduce the number of variables (generally more crucial than the number of polynomials). Other approaches could be maybe investigated. In our experiments, Grœbner basis of intersections of ideals are computed, i.e. $I \cap J$ where J is the ideal of all the polynomials not depending on $x_1, \ldots, x_\kappa$. It may seem like overkill in the sense we only need one equation in the form

$$\phi(\boldsymbol{c}, x, S, \mathsf{coeff}) = 0$$

7.3 Efficiency

As mentioned above, the running time of the homomorphic operators is polynomial as long as the expanded representation of the polynomials ν_ℓ and η_ℓ is polynomial-size. In all the experiments conducted in this section, the polynomials ν_ℓ and η_ℓ are linear. Under this choice, the running time of Add is $O(\kappa^5)$ and the running time of Mult is $O(\kappa^6)$.

Let us consider $\kappa = 13$ (ensuring the inefficiency of the basic attack[11]). Evaluating Add consists of evaluating 26 degree-4 variate-26 polynomials. To achieve this, 3×10^6 multiplications over $\mathbb{Z}_n$ should be achieved. Similarly, evaluating Mult requires around $26 \times (3 \times 10^6) = 78 \times 10^6$ multiplications.

It should be noticed that evaluating these operators consists of evaluating expanded polynomials. It can be thus highly parallelized. Moreover, some improvements can be imagined but this is not the purpose of this paper.

8 Future Work / Open questions

Several ways can be chosen to extend our security analysis. The most formal one would consist of formally reducing a classical cryptographic problem to the security of our scheme. Algebraic attacks based on Grœbner basis should be deeper investigated. In our opinion, it is a nice cryptanalysis challenge. Assumption 1 should be also deeper investigated. To achieve it, one could also search new attacks against the naive encryption scheme (which is relatively simple to analyze). Finally, the factoring assumption is required to make some algebraic attacks based on Grœbner basis irrelevant. Indeed, let us consider the public operator $\mathsf{Add} = (q_1, \ldots, q_{2\kappa})$. Without the factoring assumption, the rows $\boldsymbol{s}_{2\ell}$ could be efficiently recovered by solving[12] the following equation

$$\langle \boldsymbol{s}_{2\ell}, (q_1, \ldots, q_{2\kappa}) \rangle = \eta_\ell \nu_0 \mathcal{L}_{2\ell, 2\ell}$$

[10] γ is the parameter used in Section 6.3.

[11] which appears to be the most efficient identified attack

[12] by using variable elimination technics applied on the ideal of the polynomials derived from $\langle \boldsymbol{s}_{2\ell}, (q_1, \ldots, q_{2\kappa}) \rangle = \eta_\ell \nu_0 \mathcal{L}_{2\ell, 2\ell}$.

New ideas should be considered to remove this assumption. We carried on some preliminary experiments in this sense. In order to increase the ratio between the number of variables and the number of polynomials in I, we considered higher degree polynomials ν_ℓ, η_ℓ while S was chosen as follows,

$$S = \begin{pmatrix} S_1 & 0 & \cdots & 0 \\ 0 & S_2 & \cdots & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & S_\kappa \end{pmatrix}$$

where $S_1, \ldots, S_\kappa$ are square matrices. Assume $\kappa = \Theta(\lambda)$, the running times of the homomorphic operators remain polynomial as long as the size of the matrices S_i is $O(1)$ and the degree of the polynomials ν_ℓ and η_ℓ is polynomial. We obtained promising experimental results suggesting recovering S by using elimination technics (e.g. with Grœbner basis) is (highly) exponential with the degree of these polynomials. Informally speaking, this suggests that the result encapsulated by Proposition 3 remains true without the factoring assumption.

In our opinion, many other developments could be made. For instance, our scheme can be straightforwardly turned into a HE over real numbers. To achieve this, it suffices to choose S over the reals, e.g. $S \leftarrow [0,1]^{2\kappa \times 2\kappa}$. Indeed, $\boldsymbol{c}$ and $a \times \boldsymbol{c}$ are encryptions of the same value for any $a \in \mathbb{R}^*$. Moreover $\boldsymbol{c}$ and $\boldsymbol{c} + \varepsilon$ are encryptions of close values. Hence, the ciphertexts can be normalized after each homomorphic operation avoiding that ciphertext-size leaks information. We carried on promising experiments in this sense.

References

[AM09] Divesh Aggarwal and Ueli M. Maurer. Breaking RSA generically is equivalent to factoring. In *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, pages 36–53, 2009.

[Buc06] Bruno Buchberger. Bruno buchberger's phd thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. *Journal of Symbolic Computation*, 41(3):475–511, 2006. Logic, Mathematics and Computer Science: Interactions in honor of Bruno Buchberger (60th birthday).

[CGGI18] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *IACR Cryptology ePrint Archive*, 2018:421, 2018.

[CNT12] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *EUROCRYPT*, pages 446–464, 2012.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and*

Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I, pages 617–640, 2015.

[Gav16] Gérald Gavin. An efficient somewhat homomorphic encryption scheme based on factorization. Cryptology ePrint Archive, Report 2016/897, 2016. http://eprint.iacr.org/2016/897.

[GB19] Gérald Gavin and Stéphane Bonnevay. Fractional lwe: a nonlinear variant of lwe. Cryptology ePrint Archive, Report 2019/2502, 2019. http://eprint.iacr.org/2019/2502.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[GHS12a] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, pages 465–482, 2012.

[GHS12b] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the aes circuit. In *CRYPTO*, pages 850–867, 2012.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 75–92, 2013.

[GT20] Gérald Gavin and Sandrine Tainturier. New ideas to build noise-free homomorphic cryptosystems. In Abderrahmane Nitaj and Amr M. Youssef, editors, *Progress in Cryptology - AFRICACRYPT 2020 - 12th International Conference on Cryptology in Africa, Cairo, Egypt, July 20-22, 2020, Proceedings*, volume 12174 of *Lecture Notes in Computer Science*, pages 423–451. Springer, 2020.

[LNV11] Kristin Lauter, Michael Naehrig, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? *IACR Cryptology ePrint Archive*, 2011:405, 2011.

[Rot11] Ron Rothblum. *Homomorphic Encryption: From Private-Key to Public-Key*, pages 219–234. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[SS10] Damien Stehlé and Ron Steinfeld. Faster fully homomorphic encryption. In *ASIACRYPT*, pages 377–394, 2010.

[vDGHV10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *EUROCRYPT*, pages 24–43, 2010.

A Implementation of the naive **Add** in the case $\kappa = 1$

In this section, we provide an example of the implementation of the homomorphic scheme for $\kappa = 1$. Let $S = [s_{ij}] \in \mathbb{Z}_n^{2\times 2}$ and $\Delta = s_{11}s_{22} - s_{12}s_{21}$.

$\mathsf{Add} = (q_1, q_2) \leftarrow \mathsf{AddGen}(S)$ is defined by

$$\begin{aligned}\Delta \cdot q_1(\boldsymbol{u}, \boldsymbol{v}) =&(2s_{22}s_{11}s_{21} - s_{12}s_{21}^2)u_1v_1\\ &+s_{22}^2 s_{11}(u_1v_2 + u_2v_1)\\ &+s_{12}s_{22}^2 u_2v_2\end{aligned}$$

$$\begin{aligned}\Delta \cdot q_2(\boldsymbol{u}, \boldsymbol{v}) =& - s_{11}s_{21}^2 u_1v_1\\ &-s_{21}^2 s_{12}(u_1v_2 + u_2v_1)\\ &+(s_{11}s_{22}^2 - 2s_{21}s_{12}s_{22})u_2v_2\end{aligned}$$

B SageMath implementation

```
############################################################
#
# The polynomial nu_i and eta_i are linear
# kappa should be here smaller than 15
# The operators Rand are not considered here
#
############################################################

import time
n=random_prime(2^80,true)
print(n)
kappa=3
ka=2*kappa
R.<u1,u2,u3,u4,u5,u6,u7,u8,u9,u10,u11,u12,u13,u14,u15,u16,u17,u18,u19,u20,v1,v2,v3,v4,v5,v6,v7,v8,v9,
v10,v11,v12,v13,v14,v15,v16,v17,v18,v19,v20>=PolynomialRing(FiniteField(n),40)
U=[u1,u2,u3,u4,u5,u6,u7,u8,u9,u10,u11,u12,u13,u14,u15,u16,u17,u18,u19,u20]
V=[v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11,v12,v13,v14,v15,v16,v17,v18,v19,v20]

################### KeyGen(kappa) ############################

ss=list((randint(0, n) for i in range(ka*ka)))
S=matrix(FiniteField(n),ka,ka,ss)
SI=S.inverse()
u=vector(U[0:ka:1])
v=vector(V[0:ka:1])
su=S*u
sv=S*v

###################### Operator Add ############################

Sommenui=vector([0]*ka)
nu0=vector(randint(0,n) for i in range(ka))
SAdd=[]
```

```
for i in range(kappa):
    if (i==kappa-1):
        etai=vector(randint(0,n) for i in range(ka))
        SAdd.append((etai*v)*((nu0*u)*(su[2*i]*sv[2*i+1]+sv[2*i]*su[2*i+1])-(Sommenui*u)*su[2*i+1]*sv[2*i+1]))
        SAdd.append((etai*v)*(nu0*u)*su[2*i+1]*sv[2*i+1])
    else:
        nui=vector(randint(0,n) for i in range(ka))
        Sommenui=Sommenui+nui
        etai=vector(randint(0,n) for i in range(ka))
        SAdd.append((etai*v)*((nu0*u)*(su[2*i]*sv[2*i+1]+sv[2*i]*su[2*i+1])+(nui*u)*su[2*i+1]*sv[2*i+1]))
        SAdd.append((etai*v)*(nu0*u)*su[2*i+1]*sv[2*i+1])

SSAdd=vector(SAdd)
Add1=SI*SSAdd


def Add(c1,c2):
    e1=[0]*30
    e2=[0]*30
    for i in range(ka):
        e1[i]=c1[i]
        e2[i]=c2[i]
        c3=[0]*ka
        for i in range(ka):
            c3[i]=Add1[i].subs(u1=e1[0],u2=e1[1],u3=e1[2],u4=e1[3],u5=e1[4],u6=e1[5],u7=e1[6],u8=e1[7],u9=e1[8],
            u10=e1[9],u11=e1[10],u12=e1[11],u13=e1[12],u14=e1[13],u15=e1[14],u16=e1[15],u17=e1[16],u18=e1[17],
            u19=e1[18],u20=e1[19],v1=e2[0],v2=e2[1],v3=e2[2],v4=e2[3],v5=e2[4],v6=e2[5],v7=e2[6],v8=e2[7],
             v9=e2[8],v10=e2[9],v11=e2[10],v12=e2[11],v13=e2[12],v14=e2[13],v15=e2[14],v16=e2[15],v17=e2[16],
            v18=e2[17],v19=e2[18],v20=e2[19])
    return vector(c3)

######################## Operator Mult ############################

O=[]
for k in range(kappa):
    Sommenui=vector([0]*ka)
    nu0=vector(randint(0,n) for i in range(ka))
    SOk=[]
    for i in range(kappa):
        if (i==kappa-1):
            etai=vector(randint(0,n) for i in range(ka))
            SOk.append((etai*v)*((nu0*u)*(su[2*i]*sv[(2*i+2*k)
            SOk.append((etai*v)*(nu0*u)*su[2*i+1]*sv[(2*i+1+2*k)
        else:
            nui=vector(randint(0,n) for i in range(ka))
            Sommenui=Sommenui+nui
            etai=vector(randint(0,n) for i in range(ka))
            SOk.append((etai*v)*((nu0*u)*(su[2*i]*sv[(2*i+2*k)
            SOk.append((etai*v)*(nu0*u)*su[2*i+1]*sv[(2*i+1+2*k)
    SSOk=vector(SOk)
    O.append(SI*SSOk)


def Mult(c1,c2):
    e1=[0]*30
    e2=[0]*30
    for i in range(ka):
        e1[i]=c1[i]
        e2[i]=c2[i]
        c=[0]*kappa
        for k in range(kappa):
            c[k]=[0]*ka
            for i in range(ka):
                c[k][i]=O[k][i].subs(u1=e1[0],u2=e1[1],u3=e1[2],u4=e1[3],u5=e1[4],u6=e1[5],u7=e1[6],
                u8=e1[7],u9=e1[8],u10=e1[9],u11=e1[10],u12=e1[11],u13=e1[12],u14=e1[13],u15=e1[14],
                u16=e1[15],u17=e1[16],u18=e1[17],u19=e1[18],u20=e1[19],v1=e2[0],v2=e2[1],v3=e2[2],
                v4=e2[3],v5=e2[4],v6=e2[5],v7=e2[6],v8=e2[7],v9=e2[8],v10=e2[9],v11=e2[10],v12=e2[11],
                v13=e2[12],v14=e2[13],v15=e2[14],v16=e2[15],v17=e2[16],v18=e2[17],v19=e2[18],v20=e2[19])
```

```
    c12=c[0]
    for i in range(1,kappa):
        c12=Add(c12,c[i])
    return c12

###################### Encrypt/decrypt ##########################

def Encrypt(x):
    xx=vector(randint(0,n) for i in range(kappa))
    Sumxx=0
    for i in range(kappa-1):
        Sumxx=Sumxx+xx[i]
    xx[kappa-1]=n-Sumxx
    rr=vector(randint(1,n) for i in range(kappa))
    Sc=[]
    for i in range(kappa):
        Sc.append(rr[i]*xx[i])
    Sc.append(rr[i])
    SSc=vector(Sc)
    return SI*SSc


def Decrypt(c):
    Sc=S*c
    x=0
    for i in range(kappa):
        x=x+Sc[2*i]/Sc[2*i+1]
    return x

######################## Main ############################

tp1=time.clock()
print('ca commence...')
c1=Encrypt(9)
c2=Encrypt(11)
print(Decrypt(Add(c1,c2)))
print(Decrypt(Mult(c1,c1)))
tp2=time.clock()
print('running time: ',tp2-tp1)
```

C Algebraic attack dealing with $\kappa = 2$

```
n=97
ka=4
R. <d11, d12, d13, d14, d21, d22, d23, d24, e11, e12, e13, e14, e21, e22, e23, e24, f11, f12, f13, f14,
f21, f22, f23, f24, x1, x2, c11, c12, c13, c14, c21, c22, c23, c24, a11, a12, a13,a14,a21,a22,a23, a24,
b11, b12, b13, b14, b21, b22, b23, b24, u1, u2, u3, u4, v1, v2, v3, v4, s11, s12, s13, s14, s21, s22,
s23, s24, s31, s32, s33, s34, s41, s42, s43, s44, R1>=PolynomialRing(FiniteField(n),75)
alpha=list();qq=list();pi=list();L=list();LL=list();ZL=list()
S = matrix(4, 4, [s11,s12,s13,s14,s21,s22,s23,s24,s31,s32,s33,s34,s41,s42,s43,s44])
u=vector([u1,u2,u3,u4])
v=vector([v1,v2,v3,v4])
a1=vector([a11,a12,a13,a14]); a2=vector([a21,a22,a23,a24])
b1=vector([b11,b12,b13,b14]); b2=vector([b21,b22,b23,b24])
c1=vector([c11,c12,c13,c14]); c2=vector([c21,c22,c23,c24])
d1=vector([d11,d12,d13,d14]); d2=vector([d21,d22,d23,d24])
e1=vector([e11,e12,e13,e14]); e2=vector([e21,e22,e23,e24])
f1=vector([f11,f12,f13,f14]); f2=vector([f21,f22,f23,f24])
aa1=vector([randint(1, n) for i in range(4)]);aa2=vector([randint(1, n) for i in range(4)])
bb1=vector([randint(1, n) for i in range(4)]);bb2=vector([randint(1, n) for i in range(4)])
cc1=vector([randint(1, n) for i in range(4)]);cc2=vector([randint(1, n) for i in range(4)])
dd1=vector([randint(1, n) for i in range(4)]);dd2=vector([randint(1, n) for i in range(4)])
ee1=vector([randint(1, n) for i in range(4)]);ee2=vector([randint(1, n) for i in range(4)])
```

```
ff1=vector([randint(1, n) for i in range(4)]);ff2=vector([randint(1, n) for i in range(4)])
SS=matrix(FiniteField(n),4,4,(randint(1,n) for i in range(16)))
TT=SS.inverse()
c=[1,2,3,4]
alpha.append(c[0]*s11+c[1]*s12+c[2]*s13+c[3]*s14-x1*(c[0]*s21+c[1]*s22+c[2]*s23+c[3]*s24))
alpha.append(c[0]*s31+c[1]*s32+c[2]*s33+c[3]*s34-x2*(c[0]*s41+c[1]*s42+c[2]*s43+c[3]*s44))
su=S*u
sv=S*v
ssu=SS*u
ssv=SS*v
O=vector([1,1,1,1])

L.append(vector([(d1*v) * ((O*u) * (su[0]*sv[0]) + (a2*u) * (su[1]*sv[1])), (d1*v) * (O*u) *su[1]*sv[1],
(d2*v) * ((O*u) * (su[2]*sv[2]) - (a2*u) * (su[3]*sv[3])), (d2*v) * (O*u) * su[3]*sv[3]]))

L.append(vector([(e1*v) * ((O*u) * (su[0]*sv[1] + su[1]*sv[0]) + (b2*u) * (su[1]*sv[1])),(e1*v) *
(O*u)*su[1]*sv[1], (e2*v) * ((O*u) * (su[2]*sv[3] + su[3]*sv[2]) - (b2*u) * (su[3]*sv[3])),(e2*v) *
(O*u)*su[3]*sv[3]]))

L.append(vector([(f1*v) * ((O*u) * (su[0]*sv[2]) + (c2*u) * (su[1]*sv[3])), (f1*v) * (O*u) * su[1]*sv[3],(f2*v)
* ((O*u) * (su[2]*sv[0]) - (c2*u) * su[3]*sv[1]), (f2*v) * (O*u) * su[3]*sv[1]]))

LL.append(vector([(dd1*v) * ((O*u) * (ssu[0]*ssv[0]) + (aa2*u) * (ssu[1]*ssv[1])), (dd1*v) * (O*u)
* ssu[1]*ssv[1], (dd2*v) * ((O*u) * (ssu[2]*ssv[2]) - (aa2*u) * (ssu[3]*ssv[3])),(dd2*v) * (O*u) *
ssu[3]*ssv[3]]))

LL.append(vector([(ee1*v) * ((O*u) * (ssu[0]*ssv[1] + ssu[1]*ssv[0]) + (bb2*u) * (ssu[1]*ssv[1])),
(ee1*v) * (O*u) * ssu[1]*ssv[1],(ee2*v) * ((O*u) * (ssu[2]*ssv[3] + ssu[3]*ssv[2]) - (bb2*u) * (ssu[3]*ssv[3])),(ee2*v)
* (O*u) * ssu[3]*ssv[3]]))

LL.append(vector([(ff1*v) * ((O*u) * (ssu[0]*ssv[2]) + (cc2*u) * (ssu[1]*ssv[3])),(ff1*v) * (O*u) *
ssu[1]*ssv[3],(ff2*v) * ((O*u) * (ssu[2]*ssv[0]) - (cc2*u) * (ssu[3]*ssv[1])), (ff2*v) * (O*u) * ssu[3]*ssv[1]]))

for i in range(3):
    q=TT*LL[i]
    for j in range(ka):
        qq.append(q[j])

for t in range(3):
    for i in range(ka):
        for j in range(ka-i):
            u=[0]*ka
            u[i]=u[i]+1
            u[i+j]=u[i+j]+1
            for k in range(ka):
                for l in range(ka-k):
                    v=[0]*ka
                    v[k+l]=v[k+l]+1
                    v[k]=v[k]+1
                    ZL=[0]*ka
                    for mm in range(ka):
                        ZL[mm]=qq[ka*t+mm].coefficient(u1:u[0],u2:u[1],u3:u[2],u4:u[3],v1:v[0],v2:v[1],v3:v[2],v4:v[3])
                    Z=vector(ZL)
                    Y=S*Z
                    for m in range(ka):
                        p=Y[m]-L[t][m].coefficient(u1:u[0],u2:u[1],u3:u[2],u4:u[3],v1:v[0],v2:v[1],v3:v[2],v4:v[3])
                        alpha.append(p)
pi=[x1+x2-R1]
IF=R.ideal(alpha+pi)
H=IF.elimination_ideal([x1,x2])
print(H)
```