Homomorphic Encryption

Bryan Tapley
Point Loma Nazarene University
San Diego, CA

May 3, 2019

Abstract

A homomorphic encryption scheme built on polynomial rings is studied. The construction of irreducible polynomials in a given field is considered and implemented. Isomorphisms between two fields are found, and the method is evaluated. The scheme is ultimately implemented in SageMath, which is largely built on Python, and this is demonstrated.

Contents

1 Introduction

Encryption as a whole is built around protecting data by transforming it. It does not deal with the protection of data by preventing its interception, but instead attempts to prevent intercepted data from being usable by an attacker. The transformations have traditionally involved schemes such as symmetric key encryption, wherein a generated secret key is used in conjunction with some functions in order to map one user's intelligible data, denoted here as plaintext, into unintelligible data, denoted here as ciphertext. It is then sent to another user who has been given the key, which they use to transform the data back into plaintext, thereby decrypting it. In order for this key to be given to each user that needs it, asymmetric encryption is often used. This scheme is generally operationally expensive, but is used even by some applications that require speed in order to pass the key for the faster symmetric encryption scheme so that it can then be used instead. It allows a user to decrypt data encrypted by a public key they have provided to others using a secret private key that they have generated. Others can likewise decrypt data encrypted by their private key. Therefore, this scheme used by two users can allow each to encrpypt data with their respective public keys while decrypting with their individually generated and secret private keys. This, then, is how the key for symmetric encryption can be generated by one user and passed relatively safely to the other. All of this is done in order to allow each user to send ciphertext to another user who can then use it as plaintext.

Homomorphic encryption, unlike those forms of encryption discussed above, is a scheme that aims to allow the recipient of encrypted data to perform defined operations on the data without first decrypting it. These operations must then return a correct result when decrypted by the original user. As such, homomorphic encryption can be potentially useful in applications in such areas as cloud computing, where sensitive data could be encrypted and sent to servers where they can be operated on securely. A good homomorphic encryption implementation would therefore require that the cost of encryption, decryption, and operating on encrpyted data be offset by the computational speed provided by the computers the data was sent to.

In this paper, a relatively basic level of a homomorphic encryption scheme will be explored and implemented. It will allow for two operations on the encrypted data to be performed correctly. Some of the underlying principles in the method used here will be shown, and the method itself will be implemented in software.

2 Mathematical Background

2.1 Sets, Relations, and Mappings

Definition 1. *A **set** will be used to describe a well defined collection of distinct elements.*

Definition 2. *An **element** of a set is a defined entity that is distinct from other elements under their definition. It is the basic component of the set within which it is held. Together, elements form the set as a whole.*

Definition 3. *A set A being **well defined** implies that any given element is either in A or not in A. An element is in A or not in A.*

Definition 4. *The **empty set**, denoted here by ∅, is the set that contains no elements. ([Fraleigh] page 1)*

If a is an element of a given set A, then this fact can be denoted by $a \in A$. Each set will be described by listing its elements in braces or by placing an element that can build a set with the property describing this construction in braces. For instance, let $\mathbb{Z}$ be the set of integers. If we want to describe the set of all even integers we use the notation

$$\{2a \,|\, a \in \mathbb{Z}\}.$$

There are several interpretations through which sets can be understood, but here they will simply be understood through definition 1. In this paper, they will primarily describe a finite or infinite number of integers, polynomials, or elements in a given algebraic structure with a given property, though they may also be used to describe some other well-defined collections under related assumptions.

Definition 5. *A nonempty **subset** B of a given set A is a set B such that if b is an element of B then b is an element of A. If B is a subset of A we will write $B \subseteq A$.*

Definition 6. *Two sets A and B are equal if and only if $B \subseteq A$ and $A \subseteq B$.*

Definition 7. *An **improper subset** of a given set A is a subset B such that $A = B$.*

Definition 8. *A **proper subset** of a given set A is a subset B such that $A \neq B$. It is represented by $B \subset A$. ([Fraleigh] page 2)*

Definition 9. *The **Cartesian product** of two sets A and B, written as $A \times B$, is the set*

$$A \times B = \{(a, b) \,|\, a \in A, b \in B\}.$$

*Likewise, the **cartesian product** of sets $A_1, A_2, A_3, \ldots, A_n$ where $n \in \mathbb{Z}^+$ is the set*

$$A_1 \times A_2 \times A_3 \times ... \times A_n = \{(a_1, a_2, a_3, ..., a_n) \,|\, a_i \in A_i \text{ where } i \in \mathbb{Z},\ 1 \leq i \leq n\}$$

Example 10. *Let $A = \{1, 2, 3\}$, $B = \{1, 2\}$, $C = \{3, 4\}$. Then*

$$\begin{aligned} A \times B \times C \;=\; & \{(1,1,3),(1,1,4),(1,2,3),(1,2,4),(2,1,3),(2,1,4), \\ & (2,2,3),(2,2,4),(3,1,3),(3,1,4),(3,2,3),(3,2,4)\}. \end{aligned}$$

[Fraleigh] page 3

Definition 11. *A **relation** ϕ between two sets A and B is a subset of $A \times B$. If $(a, b) \in \phi$ we say that a is related to b. ([Fraleigh] page 3)*

Definition 12. *A **mapping** or **function** ϕ of a set A into a set B is a relation ϕ relating each $a \in A$ into exactly one b in B. That is, if $(a, b_1) \in \phi$ and $(a, b_2) \in \phi$ then $b_1 = b_2$. The notation $\phi : A \to B$ denotes a function ϕ from a set A to a set B. If $(a, b) \in \phi$ we write $\phi(a) = b$ instead. ([Fraleigh] page 4)*

Definition 13. *A function $\phi : A \to B$ is **one to one** if and only if $\phi(a_0) = \phi(a_1)$ then $a_0 = a_1$ where $a_0, a_1 \in A$. ([Fraleigh] page 4)*

Definition 14. *A function $\phi : A \to B$ is **onto** if for all $b \in B$ there exists $a \in A$ such that $\phi(a) = b$. ([Fraleigh] page 4)*

This, then, describes a function that maps each element in A onto every element of B. However, it could map multiple elements in A onto the same element in B given that it is not specified as a one to one function.

Definition 15. *A function $\phi : A \to B$ is **bijective** if it is both one to one and onto.*

Bijective functions map each element in A into exactly one element in B, but they also map onto each element of B. As such, if a bijection exists between two sets A and B we say that the sets A and B have an equivalent number of elements.

Definition 16. *The **cardinality** of a set A is the number of elements of A. It is generally denoted by $|A|$. As noted above, a bijection between sets A and B means that A and B have the same cardinality. ([Fraleigh] page 5)*

Definition 17. *If a function $\phi : A \to B$ is bijective, there exists an **inverse function** $\phi^{-1} : B \to A$ that is also bijective, such that $\phi^{-1}(b) = a$ if and only if if $\phi(a) = b$. It is a relation that swaps the ordered pairs, that is, $(a, b) \in \phi$ if and only if $(b, a) \in \phi^{-1}$.*

2.2 Binary Operations

Definition 18. *A **binary operation** on a given set A is a function denoted here by $*$ such that $A \times A$ is mapped into A by $*$.*

$$* : A \times A \to A.$$

We write $*((a_0, a_1)) = a_0 * a_1 = a_2$ ([Fraleigh] page 20)

A binary operation therefore maps each (a_0, a_1) in A to a particular element a_2 in A by a given rule. As such, it is closed under this rule since applying a binary operation on a set A to any two elements in A will result in a single element that is also in A. Further, it is a relation that relates an element of $A \times A$ to an element of A, so its result is an element in $(A \times A) \times A$.

Definition 19. *A binary operation $*$ is **associative** if and only if $(a_0 * a_1) * a_2 = a_0 * (a_1 * a_2)$ for all a_0, a_1, a_2 in A. ([Fraleigh] page 23)*

Example 20. *If S is the set of integers, then addition is an associative binary operation on S since any two integers added together give an integer and for any integers a, b, and c it is the case that $(a + b) + c = a + (b + c)$.*

Definition 21. *A binary operation $*$ on a set A is **commutative** if and only if $a_0 * a_1 = a_1 * a_0$ for all a_0, a_1 in A ([Fraleigh] page 23)*

2.3 Equivalence Relations and Partitions

Definition 22. *An **equivalence relation** $\sim$ is a relation on a given set A such that these three properties hold:*

1. *The Reflexive Property: $a \sim a$ for all a in A.*
2. *The Symmetric Property: if $a \sim b$ then $b \sim a$ for all a and b in A.*
3. *The Transitive Property: if $a \sim b$ and $b \sim c$ then $a \sim c$ for all a, b and c in A.*

([Fraleigh] page 7)

Definition 23. *A **partition** of a given set A maps each element of A into exactly one disjoint nonempty subset of A. Each of these subsets is called a **cell**. It is therefore itself a collection of a defined group of nonempty subsets of A. ([Fraleigh] page 6)*

Definition 24. *A collection of sets $A_1, A_2, ..., A_N$ where $n \in \mathbb{Z}^+$ is **disjoint** if for each pair A_i, A_j where $i, j \in \mathbb{Z}$ and $1 \leq i < j \leq n$, A_i and A_j do not contain any of the same elements. ([Fraleigh] page 6)*

Theorem 25. *Each partition on a set A gives an equivalence relation denoted here by $\mathcal{R}$ on A such that for $a_0, a_1 \in A$, $(a_0, a_1) \in \mathcal{R}$ if and only if a_0 and a_1 are in the same cell of the partition.*

Proof. Given a set A, let $x, y \in A$, let p be any partition of A, and let $\sim$ be a relation such that $x \sim y$ if and only if x and y are in the same cell in p. Then x, which can be any given element in A, is in its own cell so that $x \sim x$ and the reflexive property therefore holds. Further, if $x \sim y$ then x and y are in the same cell of p by definition of $\sim$, so $y \sim x$ and the symmetric property holds. And if for $x, y, z \in A$, $x \sim y$ and $y \sim z$, then by definition of $\sim$ x and y are in the same cell in p, but y and z are likewise in the same cell in p. Since this is true, it must be the case that x and z are in the same cell in p, so by definition of $\sim$, $x \sim z$ and the transitive property holds. Therefore, since the reflexive, symmetric, and transitive properties hold, any partition p on a set A induces an equivalence relation on A. ([Fraleigh] page 8) □

Theorem 26. *Each equivalence relation $\sim$ on a nonempty set A gives a partition on the set A such that a given cell containing element $a \in A$ is of the form $\{x \in A \mid x \sim a\}$, and each partition of A gives an equivalence relation $\sim$ such that $a \sim b$ if and only if a and b are in the same cell of the partition. ([Fraleigh] page 8)*

Definition 27. *An **equivalence class** is a cell in a partition p on a given set A such that p was induced by an equivalence relation. As such, an equivalence relation on A gives a partition that is a set of equivalence classes. ([Fraleigh] page 8)*

2.4 Congruence and Division

Theorem 28. *Let $a \in \mathbb{Z}$ and $b \in \mathbb{Z}^+$. Then there exists a unique pair $q, r \in \mathbb{Z}$ such that $a = qb + r$ and$0 \leq r < b$. Here, r is called the **remainder**. ([Jones] page 2)*

Definition 29. *If $a, b \in \mathbb{Z}$ and $a = bn$ for some $n \in \mathbb{Z}$ then we say that b **divides** a. ([Jones] page 3)*

Definition 30. *Let n be a positive integer. Two integers a and b are **congruent** if it is true that $a = q_0 n + r_0$, $0 \leq r_0 < n$ and $b = q_1 n + r_1$, $0 \leq r_1 < n$ then $r_0 = r_1$. ([Jones] page 38)*

Lemma 31. *For $a, b \in \mathbb{Z}$ if $n \in \mathbb{Z}$ and $n \geq 1$ then $a \equiv b \ mod \ (n)$ if and only if n divides $a - b$ ([Jones] page 39)*

Proof. Using Definition 27 and assuming it is the first case that $a \equiv b \bmod (n)$, let $a = q_0 n + r_0$ where $q_0, r_0 \in \mathbb{Z}$ and $0 \leq r_0 < n$, and likewise let $b = q_1 n + r_1$ where $q_1, r_1 \in \mathbb{Z}$ and $0 \leq r_1 < n$. Then $a - b = (q_0 - q_1)n + (r_0 - r_1)$. But $r_0 = r_1$ by Definition 27, so $a - b = (q_0 - q_1)n$. Then, by definition of divides and by closure in $\mathbb{Z}$, n divides $(a - b)$. ([Jones] page 39) □

Lemma 32. *For a given $n \in \mathbb{Z}$ such that $n \geq 1$, these three facts hold:*

1. *$a \equiv a \ mod\ (n)$ for all a in $\mathbb{Z}$*
2. *For all $a, b \in \mathbb{Z}$, if $a \equiv b \ mod\ (n)$ then $b \equiv a \ mod\ (n)$*
3. *For all $a, b, c \in \mathbb{Z}$, if $a \equiv b \ mod\ (n)$ and $b \equiv c \ mod\ (n)$ then $a \equiv c \ mod\ (n)$.*

([Jones] page 39)

The proof of Lemma 31 follows from Lemma 30. This implies that for each $n \geq 1$ and $n \in \mathbb{Z}$ congruence over $\mathbb{Z}$ using mod (n) is an equivalence relation on $\mathbb{Z}$ as the relation satisfies the reflexive, symmetric, and transitive properties. Therefore the relations gives a partition of $\mathbb{Z}$ into n disjoint equivalence classes since each equivalence class corresponds to a particular remainder that can be given by any integer mod n. Let $\mathbb{Z}_n$ denote these equivalence classes using congruences as shown above. Later, these properties and some extensions of them to the ring of polynomials over a field will prove to be important pieces that will be required for the homomorphic encryption scheme proposed in this paper. ([Jones] page 39)

2.5 Groups

Now that relations, binary operations, maps, equivalence relations, and congruences have been explored, let groups be considered. These will later be a foundational component of the algebraic structures within which the homomorphic encryption scheme to be considered will operate.

Definition 33. *A **group** is a set G with a binary operation * such that * is associative on G, there is an identity element e in G such that for all $a \in G, e * a = a * e = a$, and there is an inverse for each element in G that is, for all $a \in G$ there exists $a' \in G$ such that $a * a' = a' * a = e$. ([Lidl] page 2)*

Definition 34. *A group G is **abelian** if for all a, b in G it is the case that $a * b = b * a$. ([Fraleigh] page 39)*

Definition 35. *A multiplicative group G is a **cyclic** group if there exists some element a in G such that for every element b in G it is the case that $a^k = b$ for some integer k. Here, a is a **generator** of G. ([Lidl] page 3)*

Definition 36. *The group formed by the equivalence classes modulo an integer n and under the binary operation $[a] + [b] = [a + b]$ where a, b are integers in their respective classes is **the group of integers modulo n**, or $\mathbb{Z}_n$. ([Lidl] page 5)*

Example 37. *Let G be the group of integers modulus 5 under addition. Then 2 is a generator and G is cyclic since we have $2^1 = 2, 2^2 = 4, 2^3 = 1, 2^4 = 3, 2^5 = 0$.*

Definition 38. *A group is **finite** if it contains a finite number of elements. The number of these elements is denoted by $|G|$ for a given group G, and this is called the group's **order**. ([Lidl] page 5)*

Definition 39. *A **subgroup** H of a group G is a subset of G that is a group under the operation of G. ([Lidl] page 6)*

Theorem 40. *Every subgroup of a cyclic group is itself a cyclic group. ([Fraleigh] page 61)*

Definition 41. *Let H be a subgroup of G. An equivalence relation $\mathcal{R}$ on G is defined by (a, b) in $\mathcal{R}$ if and only if $a = bh$ for some h in H. Each equivalence class is then denoted by $aH = \{ah : h \in H\}$.* ***Right cosets*** *are similar and denoted by $Ha = \{ha : h \in H\}$. In either case, the equivalence classes disjoint subsets of G. ([Lidl] page 6)*

Definition 42. *The* ***index*** *of H in G, where H is a subgroup of the group G, is the number of distinct left cosets that G modulo H gives if this number is finite. ([Fraleigh] page 101)*

Now, special mappings between groups will be considered, as will particular kinds of subgroups, which will each underlie important structures in the ultimate encryption scheme.

Definition 43. *Let G and H be groups with binary operations $*$ and $\cdot$ respectively. A* ***homomorphism*** *of G into H is a mapping $f : G \to H$ where $f(a * b) = f(a) \cdot f(b)$, so that f preserves the binary operation of G as it maps into H. This is an isomorphism if f is also one-to-one. ([Lidl] page 8)*

Definition 44. *The* ***kernel*** *of a homomorphism $f : G \to H$ of the group G into the group H is the set of elements of G that are mapped to the identity element of H. ([Fraleigh] page 129)*

Definition 45. *A* ***normal subgroup*** *H of a group G has equal cosets for each a in G. ([Lidl] page 9)*

Theorem 46. *Let G be a group with normal subgroup H. Then the set of left cosets of G modulo H forms a group under the operation $(aH)(bH) = (ab)H$ where a, b in G. ([Fraleigh] page 138)*

Definition 47. *Let G be a group and H be a normal subgroup of G. Then the* ***factor group*** *or* ***quotient group*** *of G modulo H is the group formed by the left cosets under the operation defined above. This is denoted by G/H. ([Fraleigh] page 139)*

Theorem 48. *Let $f : G \to f(G) = K$ be a homomorphism of a group G onto a group K. Then the kernel of f is a normal subgroup of G, and K is isomorphic to the factor group G modulo the kernel of f. Conversly, if H is any normal subgroup of G, then the mapping $g : G \to G/H$ defined by $g(a) = aH$ for a in G is a homomorphism of G onto G/H with kernel of $g = H$. ([Lidl] page 10)*

2.6 Rings

Having considered groups, we will now move to a structure that builds upon them.

Definition 49. *A* ***ring*** *is a set R with two operations, denoted here by $+$ and $*$, with these properties:*

1. *R is an abelian group with respect to $+$*
2. *$*$ is associative*
3. *The distributive laws hold so that if a, b, c are in R then $a * (b + c) = a * b + a * c$ and $(a + b) * c = a * c + a * b$.*

([Fraleigh] page 167)

If R is a ring with binary operations $+$ and $*$, it will be denoted $R(+,*)$. It follows from the above axioms and from properties of groups that the inverses of R are unique. In general, 0 will denote the the additive identity of R, and if a is in R then $a*0 = 0*a = 0$. Further, if b is in R then $a*(-b) = (-a)*b = -ab$.

Definition 50. *Much like a subgroup, a* ***subring*** *K of a ring $R(+,*)$ is a subring of R if it is a subset of R and if it's elements form a ring under the operations of R. ([Lidl] page 13)*

Definition 51. *An* ***ideal*** *J of a ring $R(+,*)$ if J is a subring of R with $a*r = r*a$ for all a in J and r in R. J is* ***principal*** *if there is an element in R that generates J. ([Lidl] page 13)*

By the definition of normal subgroups and ideals, an ideal J of a ring $R(+,*)$ is a normal subgroup of the additive group of R. As such, R is partitioned into equivalence classes by J, and these are called **residue classes** modulo J. They are denoted by $[a] = a + J$ for some a in R, and each class is really the subset formed by some element of R added to each element in J.

Definition 52. *Let J be an ideal of a ring $R(+,*)$, and let a, b in R. Then the following are binary operations:*

- $(a + J) + (b + J) = (a + b) + J$
- $(a + J) * (b + J) = (a * b) + J$

([Lidl] page 13)

Definition 53. *Let J be an ideal of a ring $R(+,*)$. A* ***residue class ring*** *is R modulo J under the two binary operations defined above, and is denoted by R/J ([Lidl] page 13)*

Definition 54. *An ideal of a ring is a* ***prime ideal*** *if it is not equal to the ring and no two elements from the ring will form an element in the ideal under the multiplicative operation unless at least one such element is also in the ideal. ([Fraleigh] page 248)*

Definition 55. *A* ***maximal ideal*** *is an ideal J of a ring R if it is not equal to the ring and if J contained in any other ideal of R, called S here, implies that $R = S$ or $J = S$. ([Fraleigh] page 247)*

Now, the Homomorphism Theorem for Rings can be built, essentially extending the Homomorphism Theorem for Groups. This moves towards building homomorphisms in the chosen encryption scheme.

Theorem 56. *Let R and K be rings. Then if ϕ is a homomorphism from R onto S the kernel of ϕ is an ideal of R and S is isomorphic to the factor ring $R/ker(\phi)$. Conversely, if J is an ideal of the ring R, then the mapping f of R onto R/J defined by $f(a) = a + J$ for a in R is a homomorphism of R onto R/J with kernel J. ([Fraleigh] page 14)*

To conclude this discussion of rings and their basic properties, some special ring properties will be noted. These will build towards an important structure, the field.

Definition 57. *A ring is a* ***ring with identity*** *if it has a multiplicative identity. Generally, this identity is denoted by 1. ([Lidl] page 11)*

Example 58. *Let $R(+,*)$ be the ring of integers modulo 7 where $+$ is addition and $*$ is multiplication. Then 1 is the multiplicative identity since for any a in R it is the case that $1*a = a*1 = a$.*

Definition 59. *A **commutative ring** is a ring with $*$ commutative, which will be multiplication in the case studied in this paper. ([Lidl] page 11)*

Example 60. *Let $R(+,*)$ be a ring of rationals number with addition and multiplication, so that each element in R is of the form x/y where x and y are integers. Then for any two elements a, b in R, the following holds: $a * b = b * a$, and R is commutative.*

Definition 61. *A ring $R(+,*)$ is an **integral domain** if it is a commutative ring with nonzero identity where $a * b = 0$ implies $a = 0$ or $b = 0$ if a, b in R. ([Lidl] page 12)*

Example 62. *Let $R(+,*)$ be a ring of the integers modulo 5 over addition and multiplication modulo 5. Then R is an integral domain. This can be determined given that if a and b are in R and $a * b = 0$ but neither a nor b are 0, it must be the case that $a * b = 5 * n$ for some n in the integers. But the nonzero elements of R are $1, 2, 3$, and 4, so the greatest common divisor of each nonzero element of R and 5 is 1. From here it can be seen that the greatest common divisor of $a * b$ and 5 must be 1, so $a * b$ modulo 5 will not be 0 unless a or b is 0.*

Example 63. *$\mathbb{Z}(+,*)$ is also an integral domain, as the integers are commutative with respect to addition and multiplication and any two integers multiplied together can only be 0 if one of the integers is itself 0.*

Definition 64. *An element a in a ring R is a **divisor** of b in R if there exists a c in R such that $ac = b$. If an element divides the identity of R then it is a **unit**. ([Fraleigh] page 172)*

Definition 65. *A ring is a **division ring** if its nonzero elements form a group under multiplication. ([Lidl] page 12)*

2.7 Fields

Fields can be understood as a ring with special properties, and its definition sheds light on this interpretation.

Definition 66. *A **field** is a commutative division ring. ([Lidl] page 12)*

Theorem 67. *The ring of residue classes of the integers modulo the principal ideal generated by a prime integer p, denoted by $\mathbb{Z}/(p)$, is a field. [Lidl] ([Lidl] page 14)*

Definition 68. *The **characteristic** of a ring R is the least integer n such that $n * r = 0$ for every r in R. It is 0 if no n exists such that this is true. ([Lidl] page 16)*

Theorem 69. *A ring not equal to $\{0\}$ with positive characteristic, an identity, and no zero divisors must have a prime characteristic. ([Lidl] page 16)*

Corollary 70. *A finite field has prime characteristic. ([Lidl] page 16)*

Theorem 71. *Let R be a commutative ring with identity. Then the following hold:*

- *An ideal J of R is maximal if and only if R/J is a field.*
- *An ideal J of R is a prime ideal if and only if R/P is an integral domain.*
- *Every maximal ideal of R is a prime ideal.*

([Lidl] page 17)

3 Polynomials

3.1 Definition

Definition 72. *A **polynomial** is a expression with any number of terms, and here polynomials with one **indeterminate**, or variable, shall be considered. Each term has some coefficient multiplied by the variable to some power. As such, if R is some ring, then a polynomial over R has the form* $f(x) = \sum_{i=0}^{n} a_i * x^i = a_0 + a_1 * x + a_2 * x^2 + ... + a_n * x^n$ *Here, n is a non-negative integer, the coefficients* a_i *are each elements of R, and x is an indeterminate. ([Lidl] page 19)*

Definition 73. *Let* $f(x) = \sum_{i=0}^{n} a_i * x^i$ *and* $g(x) = \sum_{j=0}^{m} b_j * x^j$*. Let z be the larger value of n and m.Then let the sum of these be defined as:*

$$f(x) + g(x) = \sum_{i=0}^{z} (a_i + b_i) * x^i$$

Now the product is:

$$f(x) = \sum_{k=0}^{n+m} c_k * x^k$$

Here, c_k *is a coefficient that can be found using this formula:* $c_k = \sum_{i+j=k} a_i * b_j$ *and where i is between* 0 *and n and j is between* 0 *and m. ([Lidl] page 19)*

3.2 Polynomial Rings

Definition 74. *If R is a ring and x is an indeterminate over R, then R[x] is the ring formed by all of the polynomials over R with the above binary operations of addition and multiplication. ([Lidl] page 19)*

Definition 75. *The **degree** of a polynomial is the largest exponent of the polynomial that has a nonzero coefficient. If* $f(x) = \sum_{i=0}^{n} a_i * x^i$*, then n is the degree, and this could be denoted as* $deg(f(x)) = n$ *or* $deg(f) = n$*. Further, if* $f(x)$ *is a **monic polynomial** then R has identity 1 and* $f(x)$ *has* $a_n = 1$*. ([Fraleigh] page 199)*

Theorem 76. *Let R be a ring. Then the following hold:*

- *R[x] is commutative if and only if R is commutative.*
- *R[x] is a ring with identity if and only if R has identity.*
- *R[x] is an integral domain if and only if R is an integral domain.*

([Lidl] page 20)

Since a field is a ring that is commutative, has identity, and is an integral domain, if F is a field then $F[x]$ is commutative, has identity, and is an integral domain.

Now, much like integer division there exists a division algorithm for polynomials over a field. Polynomials over a field have divisibility where g divides f if $f = g * b$ where f, g, b are all polynomials in $F[x]$ and where F is a field. This is based upon the following:

Theorem 77. *Let $g \neq 0$ be a polynomial in $F[x]$ where F is a field. Then for any f in $F[x]$ there exists polynomials q, r in $F[x]$ such that $f = q * g + r$ where $deg(r) < deg(g)$. ([Lidl] page 20)*

Theorem 78. *Let F be a field, and let the polynomials $f_1, f_2, ..., f_n$ be in $F[x]$ with at least one not 0. Then a unique monic polynomial g exists in $F[x]$ such that g divides every $f_1, f_2, ..., f_n$ and any polynomial dividing any polynomial in $f_1, f_2, ..., f_n$ divides g. g is called the **greatest common divisor** of $f_1, f_2, ..., f_n$. ([Lidl] page 21)*

Definition 79. *If $gcd(f_1, f_2, ..., f_n) = 1$ from the preceding theorem, then the polynomials listed are considered to be **relatively prime**. ([Lidl] page 21)*

Much like for the integers, the greatest common divisor of polynomials in $F[x]$ where F is a field can be computed using the Euclidean algorithm for polynomials over a field.

3.3 Irreducible Polynomials

Definition 80. *A polynomial f in $F[x]$ where F is a field is irreducible if f has a positive degree and $f = g * b$ with g, b in $F[x]$ implies g or b is a constant polynomial. That is, an irreducible polynomial allows for only trivial factorizations. ([Fraleigh] page 214)*

Example 81. *In $\mathbb{Z}(+, *)[x]$ the polynomial $x^2 + 1$ is irreducible. To demonstrate how this can change with respect to the ring the polynomial is in, for $\mathbb{Z}_2(+, *)[x]$ this same polynomial $x^2 + 1$ can be factored into $(x + 1) * (x + 1)$ since $2 * x = 0$, so it is not irreducible in this ring.*

Theorem 82. *Much like how the integers can each be expressed by a unique prime factorization, the polynomials in some $F[x]$ can each be uniquely expressed by factorization into monic irreducible polynomials. ([Lidl] page 24)*

Theorem 83. *If F is a field and f is a polynomial in $F[x]$, then the residue class ring $F[x]/(f)$ is a field if and only if f is irreducible over F. ([Lidl] page 25)*

This theorem is an integral component of the implementation developed here, as it is later used in order to construct fields from irreducible polynomials that are built over a polynomial ring.

3.4 Field Extensions

A field extension is a field F that has a collection of elements within it that themselves form a field K, so that F extends from K in that it contains the elements in K in addition to other elements. More formally, the following describes extension fields:

Definition 84. *If a field F contains a field K that is a field under the operations of F, then K is a **subfield** of F and F is a **extension field** of K ([Lidl] page 30)*

Definition 85. *A **prime subfield** of a given field F is a subfield of F that has no subfields of its own. ([Lidl] page 30)*

Definition 86. *Let θ be in a field F and K be a subfield of F. If k is a nontrivial polynomial over $K[x]$ and $K(\theta) = 0$, then θ is **algebraic** over K. ([Lidl] page 31)*

Definition 87. *Let K be a field and F an extension field of K. Let f be a polynomial in $K[x]$. If f is a product of linear factors in $F[x]$ it **splits** in F. A linear factor in $F[x]$ is of the form $x - a$ for some a in F. This collection of linear factors can have a coefficient in F. If f does split in F, then F is a **splitting field** of f over K. ([Lidl] page 34)*

Theorem 88. *If K is a field and $f(x)$ any polynomial of positive degree in $K[x]$, then there exists a splitting field of $f(x)$ over K. Any two splitting fields of $f(x)$ over K are isomorphic under an isomorphism which keeps the elements of K fixed and maps roots of $f(x)$ into each other. [Lidl] ([Lidl] page 35)*

4 Structure of Finite Fields

4.1 Elements

It can be shown that finite fields have a number of elements that adhere to a pattern. They have special properties that will allow for isomorphisms later on and that give them structural advantages in the chosen encryption scheme.

Theorem 89. *If F is a finite field it must have p^n elements, where p is a prime integer representing the characteristic of F and n is the degree of F over its prime subfield. ([Lidl] page 45)*

Lemma 90. *If F is a finite field with q elements, every element a in F satisfies the following: $a^q = a$. ([Lidl] page 45)*

Lemma 91. *Let F be a finite field with q elements and K a subfield of F. Then $x^q - x$ in the polynomial ring $K[x]$ factors in $F[x]$ and F is a splitting field of $x^q - x$ over K. ([Lidl] page 46)*

This lemma is will lead to the results of other important theorems and lemmas in the implementation discussed later on and in immediately following ideas.

Theorem 92. *For every prime p and integer n a finite field with cardinality $p^n = q$ exists. Any finite field with this cardinality is isomorphic to the splitting field of $x^q - x$ over F_p. ([Lidl] page 31)*

This theorem, in conjunction with those directly following, are what will enable the fields built in the chosen implementation to always have isomorphisms.

4.2 Irreducible Polynomials over Finite Fields

Theorem 93. *Let f be an irreducible polynomial with degree n of a polynomial ring $F_q[x]$. Then f has a root α in F_{q^m} and all of the roots of f are simple and given by m distinct elements: $\alpha, \alpha^q, \alpha^{q^2}, ..., \alpha^{q^{m-1}}$. ([Lidl] page 49)*

Corollary 94. *Any two irreducible polynomials in $F_q[x]$ with equivalent degrees have isomorphic splitting fields. ([Lidl] page 49)*

This corollary ultimately demonstrates that the splitting fields that will be constructed will be isomorphic provided that the same prime and degree are used. The problem will be finding this isomorphism, which will be considered next.

5 Homomorphic Encryption

Having discussed the structures that will ultimately be used in the chosen encryption scheme, consideration will be given to homomorphic encryption itself and the mapping of the chosen structures..

Homomorphic Encryption has been discussed previously, and it aims to allow for encrypted data to be operated on while encrypted and passed from a user to a receiver. In this research implementation, **homomorphisms** are constructed between finite fields. First, fields are built for the user and receiver using a randomly generated prime and degree. In practice, these should be chosen carefully, but small and randomly chosen numbers were used for simplicity and study. In this step, the irreducible polynomials are randomly chosen. Secondly, an isomorphism from the user's field to the receiver's field is found. Finally, an inverse function, which is itself an isomorphism, is found from the receiver's field to the user's field. In this scheme, only the user's polynomial need be kept secret. The user is the only one with access to the mappings in both directions and is privy to all of the information.

5.1 Chosen Algorithm

The following gives the general steps for the algorithm:

1. Choose a prime p and a degree n
2. Create polynomial rings $F_p[x]$ and $F_p[y]$
3. Choose a random irreducible monic polynomial of degree n in each field, denoted here by $f(x)$ and $g(y)$ respectively
4. Find roots of $f(x)$ in $F_p[y]/g(y)$ and choose one at random. Lift it to a polynomial in this field, denoted by $\phi(y)$.
5. Find the unique polynomial $\psi(x)$ in $F_p[x]/f(x)$ such that the greatest common divisor of $\psi(\phi(y)) - x$ and $f(x)$ is $f(x)$.
6. Mapping x to $\phi(y)$ and y to $\psi(x)$ produces the desired isomorphism and inverse isomorphism between the two fields. [Doröz, Hoffstein, Pipher, Silverman, Sunar, Whyte, and Zhang]

([Doröz, Hoffstein, Pipher, Silverman, Sunar, Whyte, and Zhang])

Using this algorithm, only $f(x)$ must be kept secure, and it is chosen randomly. In fact, there is approximately a $1/n$ chance of choosing an irreducible polynomial in the fields described above, where n is the degree of the polynomial. As such, sufficiently large primes and degrees render the capability of a brute force attack to succeed minimal. In particular, in $F_p[X]$, the number of monic irreducible polynomials with degree n is $1/n * \sum_{d|n} \Phi(n/d) * q^d$ ([Lidl] page 86). Here, $d|n$ refers to the total number of positive divisors of n, and Φ is the Moebius function, defined below.

Definition 95. *The **Moebius Function** is defined on the natural numbers so that the following holds:*

1. $\Phi(n) = 1$ *if* $n = 1$

2. $\Phi(n) = (-1)^k$ *if* n *is a product of* k *distinct primes.*

3. $\Phi(n) = 0$ *if* n *is divisible by the square of a prime*

([Lidl] page 85)

Example 96. *1. Let* $p = 5$ *and* $n = 3$.

2. *Use* $F_p[x]$ *and* $F_p[y]$ *as polynomial rings*

3. *Choose random monic irreducible polynomials, simulated here by* $f(x) = x^3 + 3x^2 + 2x + 2$ *and* $g(y) = y^3 + 4y^2 + y + 2$.

4. *A root of* $f(x)$ *in* $F_p[y]/g(y)$ *is* $2y^2 + 4y$.

5. *It can be determined that* $2x^2 + 3a$ *satisfies the requirement that the greatest common divisor of* $\psi(\phi(y)) - x$ *and* $f(x)$ *is* $f(x)$.

6. *Take* $\phi : F_p[x]/f(x) \to F_p[y]/g(y)$ *to be* $\phi(x) = 2y^2 + 4y$. *Then* $\phi^{-1} : F_p[y]/g(y) \to F_p[x]/f(x)$ *is* $\phi^{-1}(y) = 2x^2 + 3a$

Any elements in the user's field, $F_p[x]/f(x)$*, can be mapped via* ϕ *to the receiver's field. Further, it is the case that the addition and multiplication of any number of elements mapped to the reciever's field and returned via the inverse function will give the same result that the performed addition and multiplication would have given in the user's field.*

6 Sage Code

SageMath is a language built primarily with python and cython. It integrates several mathematics software packages, such as PARI, and is the language we used in our development. It can be implemented in python scripts, but this is not necessary.

The project was implemented in SageMath. PARI was used through Sage, and several components of the algorithm defined above were optimized using predefined functions and structures in these languages. The above algorithm was followed in code, and tests were performed to check its accuracy. Sage allows for quotient rings to be built from polynomial rings, and it also provides methods for most of the operations required for the chosen algorithm.

The code for the user is below. It creates a field and has a method to create the mapping to the receiver's field, as well as a method to create the inverse mapping.

```
from sage.libs.pari.convert_sage import gen_to_sage

class User:
    #Constructor sets the prime, degree to use, and builds the field
    def __init__(self, prime, deg):
        self.prime = prime
        self.deg = deg
        #build the user's field
        self.ring = PolynomialRing(GF(prime), 'a')
        self.poly = self.ring.irreducible_element(self.deg, algorithm = "random")
        self.field = QuotientRing(self.ring, self.poly, 'a')
```

```
    self.a = self.field.gen()

#Map takes in a field from the receiver and the irreducible polynomial of
#that field. The method then builds an isomorphism to the receiver's field
def FindIsomorphism(self, otherfield, g):
    kfsol= pari.polrootsff(self.poly, self.prime, g)
    self.phitemp=gen_to_sage(lift(kfsol[randint(0, self.deg-1)]),
         {'a': self.a, 'b': otherfield.gen()})

    #build a homomorphism from Kf to Pg with phitemp as the homomorphism.
    self.phi=(Hom(self.field, otherfield))([self.phitemp])
    print self.phi

#Find the inverse map from the receiver to the user
def FindInverse(self, rec):

    f = self.poly
    x = self.ring.gen()
    y = rec.ring.gen()

    #create temporary fields for finding roots
    UserTempRing = PolynomialRing(GF(rec.prime), 'v')
    uf = UserTempRing(self.poly.list())
    UserTemp = UserTempRing.quotient(uf)
    RecTempRing = PolynomialRing(GF(rec.prime), 't')
    rg = RecTempRing(rec.poly.list())
    RecTemp = RecTempRing.quotient(rg)

    #set sgsol to a vector of the roots of gt defined by ft in F_prime
    sgsol = pari.polrootsff(rg, self.prime, uf)

    #set checker to the coefficents of phitemp. Checker[0] is stored
    #on the right
    checker=self.phitemp.list()

    #run this loop as many times as totals the number of roots in sgsol
    for i in range(len(sgsol)):
        #set psitemp to the current root lifted into a polynomial with x for
        #v, y for t and convert to sage
        psitemp=gen_to_sage(lift(sgsol[i]), {'v': x, 't': y})

        #set temp to the first element in checker since psitemp^0=1
        temp=checker[0]
        #Make temp the polynomial represented by phitemp(psitemp).
        #So psitemp is plugged in for the indeterminate
        for j in range(1, len(checker)):
            temp=temp+(checker[j]*psitemp^(j))
```

```
            #if the gcd(temp-x, f)=f, the current psitemp will be an inverse
            if gcd(temp-x, f)==f:
                break

        #set psi to be the homomorphism from the receiver to the user,
        #and let that be defined by psitemp.
        self.psi=(Hom(rec.field, self.field))([psitemp])
        print self.psi

    #Map an element to the receiver
    def MapTo(self, element):
        return self.phi(element)

    #Map an element from the receiver
    def MapFrom(self, element):
        return self.psi(element)
```

Now, the code for the receiver will be shown. Like the code for the user, a field with the appropriate parameters is constructed. The receiver also has methods to perform certain operations on data that it receives. This data will have already been mapped into the receiver's field since the receiver does not have access to either of the isomorphisms.

```
from sage.libs.pari.convert_sage import gen_to_sage

#class for the receiver
class Receiver:

    #build the fields for the user and set data
    def __init__(self, prime, deg):
        self.prime = prime
        self.deg = deg

        #create receiver's field
        self.ring = PolynomialRing(GF(prime), 'b')
        self.poly = self.ring.irreducible_element(self.deg, algorithm = "random")
        self.field = QuotientRing(self.ring, self.poly, 'b')
        self.b = self.field.gen()

    #print a value in this field
    def PrintValue(self, val):
        print val

    #add two elements in this field
    def Add(self, first, sec):
        return first+sec
```

```
    #multiply two elements in this field
    def Multiply(self, first, sec):
        return first*sec

    #Takes in a list of values, adds them all in this field, and returns the value
    def AddList(self, userlist):
        value = 0*self.b
        for i in range(0, len(userlist)):
            value += userlist[i]
        return value

    #Takes in a list of values, multiplies them all in this field, and returns the
    #value
    def MultiplyList(self, userlist):
        if len(userlist) > 0:
            value = userlist[0]
            for i in range(0, len(userlist)):
                value *= userlist[i]
            return value
        else:
            return 0

    #Multiply and Add a List in this field. valist is a list of values.
    #oplist is a list of operations. valist must have one more element than oplist
    def MAList(self, valist, oplist):
        final = list()
        temp = list(valist)
        #multiply the values with a multiplication symbol, store others
        for i in range(0, len(oplist)):
            if (oplist[i] == "+"):
                final.append(temp[i])
            else:
                temp[i+1] = temp[i+1]*temp[i]

            if i == len(oplist)-1:
                final.append(temp[i+1])

        #add the remaining values
        result = final[0]
        for j in range(1, len(final)):
            result += final[j]

        return result
```

The following is the code for the controller, which is used to simulate a connection between the user's and receiver's fields. Is is also utilized to allow a user to define the fields and perform operations. These operations can be performed until the user chooses to quit.

```
load("User.sage")
load("Receiver.sage")

#main method to simulate network passing data between user and receiver
def main():
    #Get the prime and the degree from the user for now, can be changed later
    cont = 0
    while cont == 0:
        #Get the prime from the user, either input directly or as a large prime
        #greater than the input number
        choice = raw_input("Press y if you would like to input a specific prime,
        or press n to input a large number: ")
        if choice == "y":
            prime = Integer(raw_input("Enter the prime: "))
        elif choice == "n":
            intermediary = raw_input("Enter the number")
            prime = Integer(next_prime(intermediary))

        #Take in the degree
        deg = Integer(raw_input("Please enter an integer for the degree: "))

        #Create the fields for the user and receiver
        user = User(prime, deg)
        rec = Receiver(prime, deg)

        #Create the maps in each direction
        user.FindIsomorphism(rec.field, rec.poly)
        user.FindInverse(rec)

        #Ask user for operations to do until they want to quit,
        #which is performed with 8
        key = "cont"
        while key != "8":
            key = raw_input("\n\nMake a selection:\n1: Map a Value and Print
            \n2: Map a Value in Each Direction\n3: Add Two Elements
            \n4: Multiply Two Elements\n5: Add a List\n6: Multiply a List
            \n7: Add and Multiply a List\n8: Quit\n")

            #Determine whether or not to choose polynomials
            if key != "8":
                mchoice = raw_input("Select:\n1: Write polynomials for the user
                     \n2: Random: ")

            #Map a value from the user's field into the receiver's field
            if key == "1":
                #take in a polynomial
                if mchoice == "1":
```

```
        wpoly = user.field(raw_input("Write Polynomial p: "))
    #choose a random polynomial
    else:
        wpoly = user.field.random_element()
        print '{}{}'.format("Chosen Polynomial: ", wpoly)

    print '{}{}'.format("phi(p) = ", user.MapTo(wpoly))

#Map a value in the user's field into the receiver's field and back
elif key == "2":
    #take in a polynomial
    if mchoice == "1":
        wpoly = user.field(raw_input("Write Polynomial p: "))
    #choose a random polynomial
    else:
        wpoly = user.field.random_element()
        print '{}{}'.format("Chosen Polynomial: ", wpoly)

    print '{}{}'.format("phi(p) = ", user.MapTo(wpoly))
    print '{}{}'.format("psi(phi(p)) = ",
         user.MapFrom(user.MapTo(wpoly)))

#Add two elements in the user's field and in the receiver's field.
#Then map back from the receiver's field
elif key == "3":
    #take in 2 random polynomials
    if mchoice == "1":
        fpoly = user.field(raw_input("Write Polynomial p: "))
        spoly = user.field(raw_input("Write Polynomial q: "))
    #choose two random polynomials from the user's field
    else:
        fpoly = user.field.random_element()
        spoly = user.field.random_element()
        print '{}{}'.format("Chosen Polynomial p: ", fpoly)
        print '{}{}'.format("Chosen Polynomial q: ", spoly)

    print '{}{}'.format("\nResult of p+q in the user's field: ",
         fpoly+spoly)
    print '{}{}'.format("Result of phi(p)+phi(q): ",
         rec.Add(user.MapTo(fpoly), user.MapTo(spoly)))
    print '{}{}'.format("Result of psi(phi(p)+phi(q)): ",
         user.MapFrom(rec.Add(user.MapTo(fpoly),
              user.MapTo(spoly))))


#Multiply two polynomials in both fields,
#and map back from the receiver's field
```

```
elif key == "4":
    #take in two polynomials
    if mchoice == "1":
        fpoly = user.field(raw_input("Write Polynomial p: "))
        spoly = user.field(raw_input("Write Polynomial q: "))
    #choose 2 random polynomials
    else:
        fpoly = user.field.random_element()
        spoly = user.field.random_element()
        print '{}{}'.format("Chosen Polynomial p: ", fpoly)
        print '{}{}'.format("Chosen Polynomial q: ", spoly)

    #Multiply the elements in the user's field.
    #Map to the receiver's field, multiply there, and map back
    print '{}{}'.format("\nResult of p*q in the user's field: ",
         fpoly*spoly)
    print '{}{}'.format("Result of phi(p)*phi(q): ",
         rec.Multiply(user.MapTo(fpoly), user.MapTo(spoly)))
    print '{}{}'.format("Result of psi(phi(p)*phi(q)): ",
         user.MapFrom(rec.Multiply(user.MapTo(fpoly),
              user.MapTo(spoly))))

#make a list and send it to be added in the receiver's field
elif key == "5":
    length = Integer(raw_input("How many elements would you like? "))
    userval = user.field(0)
    polslist = list()
    #take in a list of polynomials in the user's field
    if mchoice == "1":
        for k in range(0, length):
            pol = user.field(raw_input("Write a Polynomial: "))
            userval += pol
            polslist.append(user.MapTo(pol))
    #choose random polynomials to add
    else:
       for j in range(0, length):
            pol = user.field.random_element()
            userval += pol
            polslist.append(user.MapTo(pol))

    #add the polynomials in the user's field.
    #Map each element to the receiver's field, add there,
    #and map back
    print '{}{}'.format("\nResult in user's field: ", userval)
    print '{}{}'.format("Result in receiver's field: ",
         rec.AddList(polslist))
    print '{}{}'.format("Result when mapped back to user: ",
```

```
          user.MapFrom(rec.AddList(polslist)))

#multiply list in the user's field.
#Map each element to the receiver's field and multiply, then map back
elif key == "6":
    length = Integer(raw_input("How many elements would you like? "))
    polslist = []
    userval = user.field(1)
    #take in a list of polynomials
    if mchoice == "1":
        for k in range(0, length):
            inpol = user.field(raw_input("Write a Polynomial: "))
            polslist.append(user.MapTo(inpol))
            userval *= inpol
    #make a list of random polynomials
    else:
       for j in range(0, length):
            rpol = user.field.random_element()
            userval *= rpol
            polslist.append(user.MapTo(rpol))

    print '{}{}'.format("\nResult of multiplication in the user's
         field:", userval)
    print '{}{}'.format("Result of multiplication in the receiver's
          field:", rec.MultiplyList(polslist))
    print '{}{}'.format("Result after mapping back to the user's
         field:", user.MapFrom(rec.MultiplyList(polslist)))

#add and multiply a list as determined by the user
elif key == "7":
    length = Integer(raw_input("How many elements would you like? "))
    polslist = list()
    ops = list()
    #take in a list of polynomials and operations
    if mchoice == "1":
        for k in range(0, length):
            inpol = user.field(raw_input("Write a Polynomial: "))
            polslist.append(user.MapTo(inpol))

            if k != length-1:
                ops.append(raw_input("Enter * for multiplication,
                     + for addition: "))
    #make a random list of polynomials and operations
    else:
       for j in range(0, length):
            availop = ["*", "+"]
            polslist.append(user.MapTo(user.field.random_element()))
```

```
                        if j != length-1:
                            ops.append(availop[randint(0, 1)])

                #print the result
                resultMAL = rec.MAList(polslist, ops)
                print '{}{}'.format("\nResult: ", resultMAL)
                print '{}{}'.format("Result in user's field: ",
                     user.MapFrom(resultMAL))

        #check if they want to go again
        cont = Integer(raw_input("Enter 0 to try another field. Enter 1 to
             quit. "))

if __name__ == "__main__":
    main()
```

7 Conclusion

Homomorphic encryption can be utilized to analyze data that must be kept private in servers apart from the original owner of the data. As such, it especially has implications in cloud computing since it could allow companies or organizations to store and analyze data through a chosen cloud service. Further, it has the potential to allow for database queries to be made privately, which is to say that it could allow a user to make a database query that retrieves their information without decrypting the query itself. Currently, homomorphic encryption schemes are too slow for practical use. The implementation discussed here suffers from this as well, but it could lead to promising avenues of research in the future. It is built on finite fields constructed from polynomial rings and irreducible polynomials, and was implemented in SageMath.

References

[Jones] Gareth A. Jones and J. Mary Jones. *Elementary Number Theory*, Springer, 2005

[Fraleigh] John B. Fraleigh. *A First Course in Abstract Algebra*, Addison-Wesley, 2003

[Biggs] Norman L. Biggs. *Codes: An Introduction to Communication and Cryptography*, Springer, London. 2008

[Lidl] Rudolph Lidl and Harald Niederreiter. *Introduction to finite fields and their applications*, Cambridge University Press, 1994.

[S$^+$09] *SageMath, the Sage Mathematics Software System (Version 8.6)*, The Sage Developers, 2019, `http://www.sagemath.org`.

[Doröz, Hoffstein, Pipher, Silverman, Sunar, Whyte, and Zhang] Yarkin Doröz, Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman, Berk Sunar, William Whyte, and Zhenfei Zhang. *Fully Homomorphic Encryption from the Finite Isomorphism Problem*.