Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Autenticación y Cifrado Basado en Ecuaciones Cuadráticas de Varias Variables

Tesis que presenta

José Luis Juan Herrera García

para obtener el Grado de

Maestro en Ciencias en Computación

Director/Codirector de la Tesis

Dr. Guillermo Morales Luna/Dr. Feliú Sagols Troncoso

México, D.F. Septiembre, 2015

Resumen

Criptosistemas asimétricos de clave pública, tales como RSA o ElGamal/ECC, podrían convertirse en esquemas inseguros cuando el cómputo cuántico sea una realidad. En 1994, Peter Shor publicó un algoritmo cuántico capaz de resolver los problemas antes mencionados en tiempo polinomial. Hay afortunadamente, otros criptosistemas que se supone son problemas intratables y resisten el ataque del algoritmo de Shor. A estos criptosistemas se les conoce como criptosistemas post-cuánticos. Uno de estos criptosistemas es el conocido como de ecuaciones cuadráticas de varias variables.

Los criptosistemas de clave pública de ecuaciones en varias variables, se basan en la dificultad de resolver un sistema de ecuaciones no lineales en un campo finito. Por ventajas de cálculo computacional, se emplean sistemas de ecuaciones cuadráticas y al problema de resolver estos sistemas cuadráticos, se le conoce como el problema de varias variables cuadrático. En este trabajo, un conjunto de polinomios cuadráticos formará la clave pública y encontrar la solución a este conjunto de polinomios, para que se cumpla con una cierta imágen deseada, es el problema intratable en el que se basan los protocolos de autenticación y cifrado planteados.

La autenticación de conocimiento nulo perfecto que proponemos, se basa en la autenticación reto/respuesta, pero es innovadora en el sentido que el verificador no adquiere ningún conocimiento nuevo cuando el probador responde al reto. Por la forma en que se construye este algoritmo de autenticación, el verificador realiza una pregunta al probador, de algo que de antemano ya conoce como respuesta y por lo tanto, el probador no revela ninguna información secreta, sólo ratifica lo que el verificador ya sabe. Por otra parte, presentamos también un algoritmo de cifrado, usando esta misma infraestructura. En este caso, Beto que quiere enviar un mensaje cifrado a Alicia, pide a ésta que encuentre los valores de las variables de los polinomios, que generan una imagen deseada de la clave pública. Cuando Alicia tiene esta solución, Beto envía a Alicia un nuevo polinomio que representa el cifrado de un bit del texto plano. Alicia que conoce la solución a los polinomios de la clave pública, podrá sustituir los valores que calculo, en el polinomio enviado, para encontrar así el bit cifrado por Beto.

Presentamos los fundamentos teóricos de las dos propuestas anteriores y los resultados obtenidos de las implementaciónes de los algoritmos para dichas propuestas. Se incluye además la forma en que se representan los polinomios públicos y el detalle de las estructuras que forman la clave privada. Estos resultados, junto con la teoría correspondiente, muestran que nuestra propuesta es factible de implementarse desde ahora, preparándonos al cómputo cuántico, cada día más cercano.

Abstract

Public key asymmetric cryptosystems, such as RSA or ElGamal/ECC, could turn into insecure schemes when quantum computing becomes a reality. In 1994, Peter Shor published a quantum algorithm capable to resolve the above problems in polynomial time. Fortunately, there are others cryptosystems, that supposedly resist Shor's algorithm. These are known, as post-quantum cryptosystems and one of them is the multivariate quadratic equations system.

Multivariate equations public key crypto systems are based on the difficulty to resolve a non-linear system of equations in a finite field. Because of computational power, quadratic equations are used and the problem to resolve these quadratic systems, is known as the multivariate quadratic problem. In the present work, the set of quadratic polynomials will be the public key and finding the values of the variables in order to satisfy a defined image, is the intractable problem that the authentication and encryption protocols presented here, are based on.

The perfect zero knowledge authentication protocol, presented in this thesis, is the well known challenge/response method, however it is innovative in the sense that the verifier does not acquire any new knowledge when the prober respond the challenge. The way this authentication algorithm is built, only allows the verifier, to receive information of something that beforehand, he already knows, therefore the prober does not disclose any secret information, he only confirms what the verifier already knows. On the other hand, we also present an encryption algorithm, using this same infrastructure. In this case, Bob who wants to send an encrypted message to Alice, ask her to find the values of the variables in the set of public polynomials in order that those polynomials comply with a certain image. When Alice finds the values of those variables, Bob sends to Alice a new polynomial, that when evaluated in the values found by Alice will reveal the encrypted bit.

We present in this thesis, the theoretical fundamentals of the two proposals mentioned above as well as the obtained results of the implemented algorithms. Public polynomials representation and details of the private key is also included. These results, along with the corresponding theory, shows that our proposals are feasible to implement now, preparing ourselves to the quantum computing era, each day closer.

Agradecimientos

Expreso mi más sincero agradecimiento, al Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional en especial al Departamento de Computación, por haberme dado cabida en ese recinto de sabiduría. Así mismo, agradezco al CONACYT por su apoyo económico durante estos dos años de estudios.

Por otra parte, agradezco al equipo administrativo del Departamento de Computación: Sofía, Felipa y Erika por su amabilidad y ayudarme con todos los trámites que tuve que realizar. También al equipo auxiliar de investigación: Lic. José Crúz, Dr. Santiago Domínguez, M.C. José Luis Flores e Ing. Arcadio Morales quienes incondicionalmente me dieron el acceso a las instalaciones y equipos necesarios para trabajar en esta tesis.

Gracias a los excelentes investigadores/profesores con que cuenta el Departamento de Computación, de cada uno de ellos aprendí de diferentes temas; especialmente al Dr. Debrup Chakraborty por su dominio avanzado de los temas que compartió conmigo en sus clases, pero especialmente por transmitirlos magistralmente. Gracias también a mis compañeros de clase, por su amistad y la ayuda que me brindaron.

A mis sinodales Dr. Luis Gerardo de la Fraga y Dr. Debrup Chakraborty les agradezco sus enriquecedores comentarios a este trabajo. A mis directores de tesis Dr. Guillermo Morales y Dr. Feliú Sagols, les agradezco que me hayan aceptado para trabajar con ellos este tema de tesis, además de todos los conocimientos que me transmitieron en el desarrollo de este trabajo y especialmente por su paciencia para conmigo.

Finalmente, a mi papá Juan, a mis hermanos Dalia y Alfonso, a mis hijos Israel y Claudia y a mi compañera Lulú, les agradezco el apoyo incondicional que me brindaron en este proyecto y en todos en los que me han acompañado.

Gracias a todos.

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1

Introducción

En septiembre del año 2013, se realizó una conferencia en Estambul, Turquía, donde se presentaron los principales problemas en Matemáticas y Ciencias de la Computación, que permanecen aún abiertos (Barreto et al., 2013). En la referencia anterior, se trata el tema de criptosistemas de clave pública de varias variables, como una alternativa que posiblemente resistirá los ataques que con computadoras cuánticas, sufrirán la factorización de enteros (Menezes et al., 1996, p. 98-99) y el problema del logaritmo discreto (Menezes et al., 1996, p. 103-113), base de RSA, ElGamal, DH y ECC.

Existen varias propuestas actualmente, para el uso de sistemas de varias variables, como se puede revisar en (Wolf, 2005): esquema Aceite-Vinagre No-Equilibrado (*UOV, Unbalanced Oil and Vinegar*), sistema triangular escalonado (*STS, Stepwise Triangular Systems*), esquema A de Matsumoto-Imai (*MIA*), ecuaciones de campo escondido (*HFE, Hidden Field Equations*). Todos estos sistemas han sufrido ataques y se han modificado para fortalecerse. El esquema Aceite-Vinagre No-Equilibrado, bajo ciertos parámetros, no ha sido roto hasta la fecha y es el que elegímos como base para la construcción de los protocolos propuestos en esta tesis.

El esquema UOV (que por su traducción a Aceite-Vinagre No-Equilibrado, llamaremos en lo sucesivo AVNE) ha sido usado básicamente para generar firmas de mensajes. Aunque también se ha propuesto usar para autenticación, en esta tesis, proponemos una opción de autenticación de conocimiento nulo perfecto, donde el verificador no adquiere ningún conocimiento adicional al que ya tiene, lo cual hace este mecanismo muy seguro. Por otra parte, para cifrado de mensajes, este esquema AVNE, presenta el problema de que el sistema de ecuaciones formado por los polinomios generados, puede tener varias soluciones y por lo tanto no se podría encontrar de forma única el descifrado de un mensaje. Sin embargo, presentamos un mecanismo de cifrado, donde la entidad que quiere cifrar un bit, generará un polinomio que sólo el receptor que posee la clave privada sabrá como resolverlo eficientemente y con ello encontrar el valor del bit enviado.

1.1 Antecedentes y motivación

El problema de factorizar un número entero es la base para la seguridad del esquema de criptografía asimétrica RSA (Rivest et al., 1978), así como el problema del logaritmo discreto da actualmente la seguridad de los esquemas basados en el logaritmo discreto, como el protocolo de intercambio de claves de Diffie-Hellman (Diffie and Hellman, 1976). Sin embargo, estos dos problemas, pueden en principio resolverse en tiempo polinomial, por medio del algoritmo de Shor (Shor, 1997) en un hipotético computador cuántico [1]. Hay varias alternativas que en principio siguen representando un problema difícil de solucionar aún con este tipo de computadores cuánticos, tales como funciones picadillo (*hash functions*), problemas en retículos (*lattices*), códigos de corrección de errores y los sistemas cuadráticos de varias variables (Barreto et al., 2013).

Dentro de este último problema, el método Aceite-Vinagre No-Equilibrado (Kipnis et al., 1999) con la parametrización adecuada, no ha sido roto hasta la fecha y consta esencialmente de un conjunto de polinomios cuadráticos en varias variables, con coeficientes en un campo $\mathbb{F}_q$. El esquema AVNE, se ha propuesto para usarse en generaciones de firmas de mensajes y también en autenticación. Es pues este esquema una opción para poder generar y solucionar polinomios que al formar un sistema de ecuaciones, se podrán resolver de manera eficiente y con ello, se logrará tener seguridad, eficiencia de cálculo y una opción a los actuales algoritmos de autenticación y cifrado.

Con este esquema de ecuaciones cuadráticas en varias variables, que como se dijo antes, es resistente a ataques que se resuelvan por algoritmos en computadores cuánticos, se tiene entonces una base sólida para sistemas criptográficos sobre la que proponemos construir dos esquemas muy usados y en constante análisis:

- Protocolo de autenticación con conocimiento nulo perfecto: en los esquemas de autenticación siempre hay algún intercambio de información entre el autenticador (*verificador*) y el autenticado (*probador*). Presentamos en esta tesis, un esquema de autenticación donde no se entregue al *verificador* ninguna información adicional a la que él ya tiene, pero que sin embargo se asegure que quien está buscando autenticarse, es alguien válido.

- Protocolo de cifrado/descifrado de mensajes: usando esta misma plataforma, vemos la opción de cifrar mensajes y obviamente contar con el descifrado de los mismos. Ya que la plataforma es segura, esto nos motivó a buscar una manera de poder tener este algoritmo de cifrado y nuestra propuesta resuelve este problema.

[1] Hasta hace algunos años computador hipotético, pero D-Wave, fabricante de computadores cuánticos tiene algunos clientes a los que ha vendido este tipo de computadores.

1.2 Descripción del problema y propuesta de solución

Como se mencionó en la Sección 1.1, buscamos desarrollar un protocolo de autenticación de conocimiento nulo y por otra parte, tener un protocolo para cifrar/descifrar mensajes. Existen desde luego protocolos para estos dos problemas, pero lo que aquí buscamos, es usar el esquema de polinomios cuadráticos en varias variables, ya que como comentamos antes, el empleo de éstos hace resistente a dichos esquemas no tan sólo a los ataques que se pueden realizar con computadores estándar, sino también por computadores cuánticos.

1.2.1 Autenticación con conocimiento nulo

Una propuesta para utilizar autenticación basada en conocimiento nulo, se encuentra en (Nachef et al., 2012). En este documento, se realiza una propuesta para el uso de polinomios en varias variables definido en un campo finito, y lo generalizan para polinomios de cualquier grado d. Sin embargo, los métodos involucran de alguna u otra manera, el enviar al *verificador*, el conjunto de soluciones a los polinomios, mismos que se obtienen por poseer la clave secreta. Desde luego, esto va enmascarado al sumarlo con algunas otras variables, de forma tal que los valores de estas soluciones, nunca están abiertamente disponibles.

Sin embargo, la propuesta anterior necesita de enviar de alguna manera, estas soluciones. Generamos entonces, una forma de poder usar conocimiento nulo para el proceso de autenticación, pero sin enviar nada relacionado con la clave secreta o bien el conjunto de soluciones generado por poseer esta clave. Es una propuesta basada solamente en lo que el *verificador* conoce y de ninguna manera trata con alguna otra información que le de datos que no ha solicitado.

Así, primeramente el *verificador* y con base en los polinomios públicos, pide al *probador*, que encuentre una solución al sistema de ecuaciones formado por los polinomios igualados a un vector que propone el *verificador*. Éste genera después, un polinomio adicional con base en una combinación lineal de la clave pública (incluyendo el vector solución) y pide al *probador* que entregue el valor al que evalúa dicho polinomio con los valores calculados de las variables de los polinomios. El *probador* auténtico podrá resolver esta pregunta de inmediato y el *verificador* sabe la respuesta, ya que fue él quien realizó la generación de dicho polinomio. De esta manera, hay nulo conocimiento enviado al *verificador* pero sólo un *probador* auténtico, puede dar la respuesta correcta con un 100 % de probabilidad.

1.2.2 Cifrado/descifrado de mensajes

Normalmente, los sistemas de polinomios cuadráticos en varias variables, no se usan para cifrar mensajes, principalmente, porque el sistema de ecuaciones que se

forma con dichos polinomios, puede tener varias soluciones y esto haría que el descifrado no fuera necesariamente único, característica que debe cumplir un sistema de cifrado.

Sin embargo, aprovechando la solución propuesta para la autenticación con conocimiento nulo, proponemos en este trabajo, un protocolo para cifrar bit a bit. Básicamente, lo que Beto debe realizar al querer enviar un bit, es generar un polinomio que evalúe al valor del bit que desea enviar. Envía este polinomios a Alicia y como ésta conoce las variables solución al sistema de ecuaciones, entonces sustituye en el polinomio recibido éstos valores, encontrando así el valor del bit enviado. Esto se repite para cada bit que Beto envíe a Alicia, hasta cifrar/descifrar todo el mensaje.

1.3 Contribuciones

Con base en el esquema AVNE que hasta la fecha tanto teórica como prácticamente ha mostrado ser seguro, hemos construido dos propuestas criptográficas que contribuyen a este campo de la manera siguiente:

1. Protocolo de autenticación con conocimiento nulo perfecto: observando la clave pública que es del conocimiento del *verificador* y considerando que éste es quien propone el vector solución que debe cumplir el sistema de ecuaciones formado por éstos polinomios, vemos que al realizar una combinación lineal de los polinomios incluyendo el vector solución, genera un nuevo polinomio que no se puede saber de donde viene ya que el campo usado es $\mathbb{F}_2$ y ahí varios términos desaparecerán al realizar esta combinación lineal, evitando dejar un rastro de los polinomios que dieron origen a éste nuevo polinomio que es el que se envía al *probador* para que de solución al mismo. Esta propuesta, contrasta con la que se menciona en (Nachef et al., 2012) ya que como comentamos antes, las variables solución, aunque enmascaradas con otros valores, se tiene que enviar al *verificador* y aunque esto puede ofrecer seguridad, en nuestra propuesta, definitivamente no se manda información adicional alguna, que el *verificador* no conozca de antemano. Esto hace al protocolo más seguro. Por otra parte, dado que la generación del polinomio que debe resolver el *probador* es una combinación lineal de algunos de los polinomios que forman la clave pública, esto no es costoso computacionalmente hablando.

2. Protocolo de cifrado/descifrado de mensajes: proponemos un protocolo que con toda seguridad puede cifrar y descifrar mensajes bit a bit. Utilizamos la misma idea del polinomio que se genera en el punto anterior, en este caso para ser resuelto por *Alicia* y que ahora genera Beto. Así, Beto genera un polinomio asegurándose que evalúe al estado del bit que se desea enviar y con esta condición satisfecha, Alicia sustituye los valores que tiene calculados como solución al sistema de ecuaciones de los polinomios públicos, para encontrar entonces el valor del bit enviado.

Ambas propuestas son novedosas y hasta donde hemos analizado, son seguras, por lo que son una aportación al campo de la criptografía. Desde luego que la comunidad criptográfica deberá criptoanalizarla y en su momento pasar como una alternativa formal.

1.4 Trabajo previo

A la fecha, la resolución de ecuaciones definidas por polinomios cuadráticos es intratable por tratarse de un problema NP-Completo (Garey and Johnson, 1990, p. 251) (Patarin and Goubin, 1997) y el conjunto de instancias resolubles es muy restringido. Debido a esto, se clasifican a los sistemas criptográficos basados en polinomios cuadráticos de varias variables como de seguridad post-cuántica.

Desde los años 80's aparecieron las primeras propuestas de esquemas criptográficos en varias variables: *MIA* en 1985, *C** en 1988, *Birational Permutation* en 1993, *HFE* en 1996, *OV* en 1997, *UOV* en 1999, *Quartz* y *Sflash* en 2001, *PMI* y *RSE(2)PKC* en 2004, *RSSE(2)PKC* y *Enhanced TTS* en 2005, *rainbow* en 2005, *MFE* en 2006, *MQQ* en 2008, *Enhanced STS* en 2010, *MQQ-Sig* y *MFE-Dio* en 2011 y *MQQ-Enc* en 2012. En (Wolf and Preneel, 2005a) se puede encontrar una clasificación y explicación de varios de estos esquemas, principalmente anteriores al año 2005.

El esquema *MQQ-Enc* (Gligoroski and Samardjiska, 2012) antes mencionado, es interesante por ser reciente y por enfocarse a cifrado de mensajes. Se trata de un esquema de cifrado probabilístico, es decir con posibles errores en el descifrado. Pertenece a la familia de sistemas cuadráticos de varias variables en cuasi-grupos (*MQQ: multivariate quadratic quasi-groups*). Tiene una función interna llamado transformación de cadenas cuasi-grupo (Gligoroski et al., 2008) donde estos cuasi-grupos se representan en forma de grupos de varias variables. Los autores declararon este esquema seguro incluso para ataques indistinguibles de elección de textos cifrados (*IND-CCA*), pero con posibles errores en el descifrado.

En el año 2013, se propuso un nuevo esquema llamado *matriz simple* o *esquema ABC* (Tao et al., 2013). Se trata de un esquema de cifrado, que usa operaciones en matrices cuyos componentes consisten de elementos en un campo finito de tamaño pequeño. Un año después (en 2014), se propuso una versión mejorada llamada *matriz simple cúbica* (Ding et al., 2014). También en el año 2014, se publicó una versión mejorada al esquema *HFE*, llamada *ZHFE* (Porras et al., 2014). Hasta el momento, no tenemos noticias de que se hayan atacado exitosamente alguno de estos esquemas. Aunque el esquema *ABC* se introdujo en el año 2013, se mejoró en el año 2014, como se dijo antes, siendo entonces los esquemas de *matriz simple cúbica* y *ZHFE* los más recientes. Esperamos su madurez, para poder trabajar con ellos.

Como se puede ver, muchas propuestas se han realizado, pero así mismo, han surgido los criptoanálisis de estos esquemas, que los han vuelto inseguros. De los mencionados anteriormente, siguen vigentes hasta hoy, por su seguridad, el esquema *AVNE* (en éste esta basado el desarrollo de esta tesis) y *rainbow*, que es una variante de *AVNE*.

Antes del esquema *AVNE*, presentaron a *OV*, es decir un esquema Aceite-Vinagre Equilibrado (*Balanced Oil Vinegar*), pero el mismo fue atacado exitosamente en 1998 como se muestra en (Kipnis and Shamir, 1998). En ese documento se muestran dos ataques algebraicos, que pueden separar eficientemente las variables aceite y vinagre y de esta manera falsificar firmas. Después de esto, se modificó este esquema y surgió *AVNE* (Kipnis et al., 1999) que siempre y cuando se cumpla con la relación de variables aceite-vinagre que veremos más adelante en ese documento, el esquema es seguro tal como ha permanecido todos estos años.

Una opción adicional a los ataques algebraicos para los esquemas criptográficos basados en polinomios cuadráticos de varias variables, es el usar bases de Gröbner (algoritmo original debido a Buchberger en 1965) para solucionar el sistema de ecuaciones que se plantean o bien, alguno de los algoritmos que han buscado mejorar la eficiencia del algoritmo de Buchberger, tales como *F4* (Faugère, 1999), *XL* (Courtois et al., 2000), *F5* (Faugère, 2002), o bien una modificación a *F5* y *G2V* (Gao et al., 2010). Sin embargo, estos algoritmos toman tiempo y espacio exponencial en el tamaño del sistema de polinomios, como se verá en la sección 2.3.6, por lo que no es factible recurrir a estos algoritmos para resolver el sistema de ecuaciones que se generan en *AVNE*.

En relación a protocolos de conocimiento nulo usando polinomios de varias variables, tenemos que en el año 2011 se publicó en (Sakumoto et al., 2011) la propuesta de un esquema de identificación de clave pública, basada en la conjetura de lo intratable de los polinomios cuadráticos de varias variables, bajo la suposición de la existencia de un esquema de compromiso no interactivo. Esto es, el *probador* con base en su clave secreta, en la clave pública y en unos datos aleatorios, genera por medio de una función de caracteres de compromiso (*String Commitment Function*[2]) tres cadenas de caracteres. Éstas las envía al *verificador* y éste pide al *probador* le envíe ahora algunos datos previamente definidos, para verificar alguna de las tres cadenas compromiso recibidas. Éste las envía y el *verificador* realiza las comprobaciones necesarias y si son lo que esperaba, acepta al *probador* de otra forma lo rechaza.

Con base en el esquema anterior, en (Nachef et al., 2012) se propusieron dos métodos para construir estos esquemas con base en polinomios de grado no exclusivamente cuadráticos, sino en general de grado d. Uno de los métodos es óptimo en relación a los cálculos que debe realizar y el otro esquema es óptimo en relación al número de bits de intercambio que genera para este esquema. Esto aplica tanto para polinomios dispersos como densos.

1.5 Organización de la tesis

Considerando este capítulo introductorio, esta tesis consta de seis capítulos. En el Capítulo 2 realizamos una revisión a varias de las consideraciones teóricas necesarias para poder homogenizar el lenguaje y criterios de cualquier lector de esta tesis. Se

[2]Esto se pude realizar por una función picadillo resistente a colisiones

tratan los temas de instancias 3SAT, polinomios cuadráticos, bases de Gröbner y las pruebas de conocimiento nulo.

En el Capítulo 3, revisamos a detalle la base sobre la que se construyó tanto el protocolo de autenticación como el de cifrado. La seguridad de este esquema, conocido como Aceite-Vinagre No-equilibrado, cuando se configura adecuadamente, no ha sido rota. Realizamos el estudio de este método y se presentamos las demostraciones necesarias para confiar en la seguridad que ofrece.

El Capítulo 4 es la propuesta que hacemos en esta tesis. Describimos los dos protocolos criptográficos que proponemos: el protocolo de autenticación con conocimiento nulo perfecto y el protocolo de cifrado. Realizamos una descripción detallada de cada protocolo y tocamos el tema de seguridad que éstos ofrecen.

En el Capítulo 5 describimos detalladamente los resultados obtenidos, tanto para el protocolo de autenticación, como para el de cifrado. Realizamos los comentarios de forma general de los programas desarrollados para lograr que realicen las funciones esperadas y describimos a detalle un ejemplo completo tanto de la autenticación, como del cifrado.

Finalmente, en el Capítulo 6 damos las principales conclusiones de este trabajo y se detallan varios de los aspectos que quedan pendientes o bien que se deben optimizar de los dos temas propuestos en esta tesis.

Los programas fuente desarrollados, los binarios generados por la compilación de estos fuentes, así como los datos necesarios para correr varios de estos programas, se encuentran en:

`http://computacion.cs.cinvestav.mx/~jherrera/AutYCif`.

En esta liga, aparecerá en la parte superior, un enlace a `Descargar estructura completa de directorios con programas fuente, compilados y datos` que representa un archivo comprimido. Será necesario descargar dicho archivo y descompactarlo, para generar una serie de directorios a los que se hará referencia a lo largo de este trabajo. En caso que no se desee descargar el archivo antes mencionado, se podrán consultar en línea los programas fuente y archivos de datos que se mencionan en el presente documento. Solamente los programas ejecutables se tendrán que descargar para poder ejecutarse. Por otra parte, los archivos de datos no mencionados explícitamente en esta tesis, habrá que descompactarlos, ya que no se dejaron de forma individual, para ahorrar espacio. Para esto, se puede accesar a la liga antes mencionada y escoger el enlace `Contenido` para tener acceso a la estructura de directorios de programas y datos. A lo largo de este documento, daremos acceso a estos directorios por medio del enlace directo Programas, aunque se podrá tener acceso también, por medio del enlace general arriba mencionado.

Capítulo 2

Conceptos preliminares

La criptografía de clave pública, usada ampliamente en nuestra época, abarca varios campos, como el firmado de mensajes y el cifrado de los mismos. Esta criptografía convencional de clave pública, se fundamenta básicamente en dos problemas: la factorización de enteros (RSA, para cifrado/descifrado de mensajes) y el problema del logaritmo discreto (DH para intercambio de claves; ElGamal, ECC para cifrado/descifrado de mensajes). Sin embargo, el hecho de que cada día tengamos más cerca la posibilidad de tener computadores cuánticos, se pone en riesgo el uso de los esquemas que usan la factorización de enteros y el logaritmo discreto:

1. En 1994, Shor (Shor, 1997) propuso un algoritmo para resolver el problema de factorización y el del logaritmo discreto, en tiempo polinomial, empleando un computador cuántico.

2. En 2001, se empleó el algoritmo de Shor en un computador cuántico implementado con resonancia magnética nuclear, para resolver la factorización del número 15. Este computador tenía 7 qubits.

3. El 28 de Noviembre de 2014, se anuncia por el sitio de Internet de noticias en ciencia, investigación y tecnología http://phys.org/ la factorización del número 56,153 por un dispositivo cuántico: *New largest number factored on a quantum device is 56,153*

4. En 2013, la empresa *D-Wave Systems* anunció un computador cuántico como el más avanzado del mundo, que cuenta con 512 qubits. Su uso principal: optimización, aprendizaje máquina, reconocimiento de patrones y detección de anomalias, análisis financiero, así como verificación y validación de hardware/software. Aunque esta empresa busca integrar nuevos descubrimientos en física, ingeniería, fabricación de computadores y ciencias de la computación, no ha anunciado explícitamente como los avances que ha logrado, han influido en el campo de la criptografía.

Pensamos sin embargo, que se acerca más cada día, el hecho de poder contar con un computador que resuelva entonces los problemas criptográficos arriba mencionados.

Considerando lo anterior, y el hecho de que en nuestros días, prácticamente cualquier dispositivo electrónico puede conectarse a Internet y que por lo tanto, dichos dispositivos requieren de usar comunicaciones seguras, nos motivó a buscar alternativas que por lo menos hasta hoy, no cuenten con un algorítmo cuántico para resolverlas en tiempo polinomial. Estas alternativas se conocen como *criptosistemas post-cuánticos* y son: retículos (*lattices*), códigos de corrección de errores, sistemas cuadráticos de varias variables y funciones picadillo (*hash functions*).

La presente tesis se basa en sistemas cuadráticos de varias variables (*multivariate quadratic systems*) y describiremos entonces cómo se forman estos polinomios y cuál es el problema a resolver cuando se trabaja con ellos en criptografía. Para ello, en este capítulo, empezaremos por realizar una revisión de 3-SAT en la Sección 2.1, ya que éste concepto, nos servirá para demostrar que un sistema cuadrático de ecuaciones es NP-Completo. Continuaremos con la forma en que se construyen los polinomios cuadráticos en varias variables en la Sección 2.2. Considerando que el ataque más básico, que se puede realizar a un esquema criptográfico cuadrático en varias variables, es resolver el sistema de ecuaciones que este representa, en la Sección 2.3 revisaremos el tema de bases de Gröbner, ya que ésta es una opción para resolver dicho sistema. Finalmente, en la Sección 2.4 revisaremos el concepto de autenticación con conocimiento nulo perfecto, para poder aplicarlo al protocolo que desarrollamos en este trabajo de tesis.

2.1 Generación de instancias 3-SAT

El problema de decisión de *satisfactibilidad booleana (SAT)* fue el primer problema NP-Completo. Si consideramos una lista de n variables booleanas $X = (X_0, \ldots, X_{n-1})$, decimos que una asignación verdadera para X es una función $t : X \to \{T, F\}$, donde T es el valor verdadero y F el falso. Así, si $t(X_j) = T$ decimos que X_j es verdadera bajo t y si $t(X_j) = F$ decimos que X_j es falso. Si X_j es una variable en X, se dice que X_j^{ε} es una *literal*, donde $\varepsilon \in \{-1, 1\}$ y $X_j^1 = X_j$ mientras que $X_j^{-1} = \bar{X}_j$ (complemento o negación de X_j). Por otra parte, una cláusula sobre X es una lista de literales sobre X representando una disyunción, por ejemplo $[X_1, X_3^{-1}, X_5, X_6]$ es una cláusula de cuatro literales que representa la disyunción de las mismas y es satisfecha por una asignación verdadera si y sólo si al menos uno de los elementos de la lista es verdadero bajo esa asignación. Así, la lista anterior será satisfecha por t *a menos* que $t(X_1) = F$, $t(X_3) = T$, $t(X_5) = F$ y $t(X_6) = F$. Una colección C de cláusulas sobre X es satisfactible si y sólo si existe alguna asignación verdadera para X que simultáneamente satisfaga todas las cláusulas en C, es decir, se tiene una conjunción de cláusulas y cada cláusula es una disyunción de literales. Tal asignación verdadera se llama *asignación verdadera satisfactora* para C. Considerando lo anterior, se puede definir ahora el problema de satisfactibilidad como:

SATISFACTIBILIDAD:
INSTANCIA: Una lista X de variables y una colección C de cláusulas sobre X.

PREGUNTA: ¿Existe una asignación verdadera satisfactoria para C?

Por ejemplo si $X = (X_0, X_1, X_2)$ y $C = [[\bar{X}_0, \bar{X}_1, X_2], [X_0, X_1, \bar{X}_2]]$ es una instancia de SAT donde sí existe una asignación verdadera que satisface a C, por ejemplo $X_0 = F, X_1 = F, X_2 = F$. Sin embargo si ahora $C = [[X_0, X_2], [X_0, \bar{X}_2], [\bar{X}_0]]$ tenemos entonces, una instancia para la que no existe una asignación verdadera que satisfaga a C.

El teorema de Cook (1971) establece: *el problema SAT es NP-Completo* (Garey and Johnson, 1990, pp. 39-44).

El problema 3-SAT es una versión restringida de SAT, donde todas las cláusulas (3-cláusulas) tienen exactamente tres literales. La demostración de que 3-SAT es también NP-Completo, se encuentra en (Garey and Johnson, 1990, p. 48-49). Las literales en estas cláusulas son diferentes tomadas por pares y como en el caso de SAT forman una disyunción. Por otra parte, una forma 3-conjuntiva (3-CF) es una lista de conjunciones de 3-cláusulas.

Veamos ahora cómo generar, de manera aleatoria, un conjunto de 3-CF, apoyados en 3-SAT y en el hipercubo.

Sea $Q_n = \{0,1\}^n$ el hipercubo n-dimensional. El conjunto de valores $Q = \{0,1\}$ se transforman en el conjunto de valores $\{-1,1\}$ por la función $\eta : \delta \mapsto \varepsilon = \eta(\delta) = 2\delta - 1$ con su inverso $\eta^{-1} : \varepsilon \mapsto \delta = \eta^{-1}(\varepsilon) = \frac{1}{2}(1+\varepsilon)$.

Por otra parte, cualquier literal determina una función booleana:

$$X_j^{\varepsilon} : Q^n \to Q, \quad x \mapsto \begin{cases} 1 & \text{si } x_j = \eta^{-1}(\varepsilon) \\ 0 & \text{de otra forma.} \end{cases}$$

Una 3-cláusula $c = [\ell_0, \ell_1, \ell_2]$ determina:

$$c : Q^n \to Q, \quad x \mapsto \begin{cases} 1 & \text{si } \exists \kappa \in \{0,1,2\} : \ell_\kappa(x) = 1 \\ 0 & \text{de otra forma.} \end{cases}$$

Y una 3-CF $f = [c_\mu]_{\mu=0}^{m-1}$ determina:

$$f : Q^n \to Q, \quad x \mapsto \begin{cases} 1 & \text{si } \forall \mu \in \{0,\ldots,m-1\} : c_\mu(x) = 1 \\ 0 & \text{de otra forma.} \end{cases}$$

Las literales, las 3-cláusulas y las 3-CF son *formas booleanas*. Una forma booleana es una contradicción si determina el mapa booleano en cero, de otra manera es satisfactible. Ahora bien, si $g : Q^n \to Q$ es una función booleana, la imagen inversa de 0, $\mathrm{Nul}(g) = g^{-1}(0)$, se conoce como el conjunto nulo de g y la imagen inversa de 1, $\mathrm{Spt}(g) = g^{-1}(1)$, se conoce como el soporte de g. Así, $\{\mathrm{Nul}(g), \mathrm{Spt}(g)\}$ es una partición del hipercubo Q^n. Entonces, una forma booleana es satisfactible si el soporte de la función booleana que ésta determina no es vacío.

El problema 3-SAT es el siguiente:

Instancia: Una 3-CF $f = [c_\mu]_{\mu=0}^{m-1}$.
Solución: La decisión de si f es satisfactible

Observamos que para cualquier 3-cláusula $c = [X_{j_0}^{\varepsilon_0}, X_{j_1}^{\varepsilon_1}, X_{j_2}^{\varepsilon_2}]$,

$$\text{Nul}(c) = \{x \in Q^n | x_{j_0} \neq \eta^{-1}(\varepsilon_0) \ \& \ x_{j_1} \neq \eta^{-1}(\varepsilon_1) \ \& \ x_{j_2} \neq \eta^{-1}(\varepsilon_2)\}$$

Es decir, los puntos en el conjunto nulo de c tienen tres componentes con valor fijo y por lo tanto $\text{Nul}(c)$ es un sub-cubo de dimensión $(n-3)$ equivalente a una subgráfica isomórfica al hipercubo Q^{n-3} de Q^n.

Nota 2.1.1 *Para cualquier* 3*-cláusula* c*,* $\text{card}(\text{Spt}(c)) = 2^n - 2^{n-3} = (2^3-1)2^{n-3}$ *y* $\text{card}(\text{Nul}(c)) = 2^{n-3}$.

Si c_0, c_1 son dos 3-cláusulas, entonces la 3-CF $f = [c_0, c_1]$ tiene como conjunto nulo $\text{Nul}(f) = \text{Nul}(c_0) \cup \text{Nul}(c_1)$, de forma tal que

$$\text{card}(\text{Nul}(f))=\text{card}(\text{Nul}(c_0))+\text{card}(\text{Nul}(c_1))\text{-card}(\text{Nul}(c_0)\cap \text{Nul}(c_1)),$$

o de manera más general, para una 3-CF $f = [c_\mu]_{\mu=0}^{m-1}$, la $\text{card}(\text{Nul}(f))$ se puede calcular por el principio de inclusión-exclusión.

Nota 2.1.2 *El cubo de* 3 *dimensiones* Q^3 *tiene* 9 *particiones, cada una formada por cuatro sub-cubos de* 1 *dimensión, i.e. cuatro aristas.*

Así, si usamos la notación _ para denotar la coordenada que varía en una arista y si representamos a las otras coordenadas por sus correspondientes valores, tenemos:

$$\begin{aligned}
e_{3,0} &= \{00_, 10_, 01_, 11_\} & e_{3,1} &= \{_00, _01, 01_, 11_\} \\
e_{3,2} &= \{00_, 10_, _11, _10\} & e_{3,3} &= \{_00, _01, _10, _11\} \\
e_{3,4} &= \{_00, _10, 0_1, 1_1\} & e_{3,5} &= \{0_0, 1_0, 0_1, 1_1\} \\
e_{3,6} &= \{0_0, 1_0, _11, _01\} & e_{3,7} &= \{0_0, 0_1, 10_, 11_\} \\
& & e_{3,8} &= \{00_, 01_, 1_0, 1_1\}
\end{aligned} \tag{2.1}$$

Cada colección $e_{3,k}$ es una partición del cubo en cuatro aristas. Sea $n > 3$. Por cada tripleta de índices $\{j_0, j_1, j_2\} \in \{0, \ldots, n-1\}^3$ y cualquier partición $e_{3,k}$ dada por las Ecuaciones 2.1, ascendemos la partición $e_{3,k}$ en una partición $e_{n,k'}$ del hipercubo n-dimensional Q^n, poniendo en las entradas numeradas por $\{i_0, i_1, i_2\}$ los patrones en $e_{3,k}$ y rellenando las otras entradas con el valor "_". Entonces, los patrones que aparecen en las particiones $e_{n,k'}$ tienen la forma $\_^{\ell_0}\delta_0\_^{\ell_1}\delta_1\_^{\ell_2}$ con $\delta_0, \delta_1 \in \{0,1\}$, $\ell_0 + \ell_1 + \ell_2 = n-2$ (ℓ_i representa el número de veces que se debe repetir el valor de relleno "_", y dado que en la partición $e_{n,k'}$ del hipercubo sólo se sustituirán dos valores dados por δ_0, δ_1 entonces restarán $n-2$ lugares a rellenar).

Propuesta 2.1.1 *En el hipercubo n-dimensional Q^n, una colección de $\sum_{i=0}^{\frac{n!}{2}-1}(2^{n-1}-i)$ particiones se pueden introducir, cada una formada por cuatro hipercubos de $(n-2)$ dimensiones, y todas estas particiones se pueden enumerar.*

Sea $n \geq 4$. Sea $\{j_0, j_1, j_2, j_3\} \in \{0, \ldots, n-1\}^4$ un 4-subconjunto del conjunto de índices $\{0, \ldots, n-1\}$ y sea $\delta_3 \in \{0, 1\}$ un bit. Para cualquier patrón $\pi = {}_{-}{}^{\ell_0}\delta_{0-}{}^{\ell_1}\delta_{1-}{}^{\ell_2}$ con los valores de δ apareciendo en las posiciones en la tripleta j_0, j_1, j_2, asociemos ahora la cláusula $c = X_{j_0}^{\varepsilon_0} \vee X_{j_1}^{\varepsilon_1} \vee X_{j_2}^{\varepsilon_2} \vee X_{j_3}^{\varepsilon_3}$ donde para cada $\nu \in \{0, 1, 2\}$, $\varepsilon_\nu = \eta(\delta)$ si $\delta \in \{0, 1\}$ aparece en la posición j_ν en π y $\varepsilon_\nu = 0$ en otro caso, donde para cada variable X, X^0 denota que la variable X no aparece en la cláusula o dicho de otra manera, tiene el valor verdadero T. Así que c es una 3-cláusula involucrando la variable X_{j_3}. Sea H el hipercubo $(n-2)$-dimensional denotado por el patrón π. Entonces, $H \cap Q^n|_{x_{j_3} \neq \delta_3}$ es un hipercubo $((n-1)-2)$-dimensional incluido en el semi-hipercubo $Q^n|_{x_{j_3} \neq \delta_3} \approx Q^{n-1}$ y en el conjunto nulo de c. Por lo tanto, el semi-hipercubo $Q^n|_{x_{j_3} \neq \delta_3}$ está incluido en el conjunto nulo de 3-CF que consiste de las cuatro 3-cláusulas asociadas de un recubrimiento del semi-hipercubo por cuatro hipercubos $((n-1)-2)$-dimensional.

Nota 2.1.3 *Cualquier subcubo $(n-1)$-dimensional de Q^n puede incluirse en el conjunto nulo de una 3-CF que consiste de cuatro 3-Cláusulas.*

Por ejemplo, consideremos $n = 6$, la 4-tupla $\{j_0, j_1, j_2, j_3\} = \{1, 2, 3, 4\}$ y el bit $\varepsilon_3 = 1$. Consideremos el recubrimiento $e_{3,6} = \{0\_0, 1\_0, \_11, \_01\}$ de las Ecuaciones 2.1. Se asciende entonces al recubrimiento de Q^6 así:

$$e_6 = \{\_0\_0\_\_, \_1\_0\_\_, \_\_11\_\_, \_\_01\_\_\}$$

ya que los valores de $e_{3,6}$ van en las posiciones j_0, j_1, j_2. La CF asociada será (para las primeras tres variables, se considera a e_6 y donde hay un 1, corresponderá a la variable negada; la última variable tiene como exponente al bit ε_3):

$$f = [X_1^1 \vee X_2^0 \vee X_3^1 \vee X_4^1, X_1^{-1} \vee X_2^0 \vee X_3^1 \vee X_4^1, X_1^0 \vee X_2^{-1} \vee X_3^{-1} \vee X_4^1, X_1^0 \vee X_2^1 \vee X_3^{-1} \vee X_4^1]$$

$$f = [X_1 \vee X_3 \vee X_4, \bar{X}_1 \vee X_3 \vee X_4, \bar{X}_2 \vee \bar{X}_3 \vee X_4, X_2 \vee \bar{X}_3 \vee X_4]$$

El hipercubo 5-dimensional $Q^6_{x_4=0} = \_\_\_\_0\_$ es un subconjunto de $\mathrm{Nul}(f)$.

Dado un punto en el hipercubo $a \in Q^n$ podríamos expresar la función característica de la mónada $\{a\}$,

$$\chi_a : \quad x \mapsto \begin{cases} 1 & \text{if } x = a \\ 0 & \text{if } x \neq a \end{cases}$$

como la función booleana determinada por una 3-CF. En otras palabras, construyamos una 3-CF f_a que tiene a a como su única asignación satisfactoria.

De esta forma, para $n = 3$, si $a = (a_0, a_1, a_2)$ y $b = (b_0, b_1, b_2) = (\eta(a_0), \eta(a_1), \eta(a_2))$ entonces:

$$f_{3,a}(X, Y, Z) = \bigwedge \left\{ X^{\eta(b_0)} \vee Y^{\eta_{b_1}} \vee Z^{\eta(b_2)} | (b_0, b_1, b_2) \in Q^3 - \{(\bar{a_0}, \bar{a_1}, \bar{a_2})\} \right\} \quad (2.2)$$

En el Algoritmo 1 resumimos los puntos antes expuestos y generamos entonces una función 3-CF que involucra cláusulas diferentes en cada corrida, ya que hay elementos aleatorios por los índices que se involucran. Generamos en total $4(n-3)+7 = 4(n-1)-1$ 3-cláusulas.

Algoritmo 1 Generación de una función en 3-CF

Entrada: Un punto a en el hipercubo n-dimensional
Salida: Una función 3-CF

1: $f_a \leftarrow [\,]$ // Inicialmente f_a es una lista vacía
2: $J \leftarrow \{0, \ldots, n-1\}$ // Inicialmente J tiene el índice completo variables
3: $n_j \leftarrow n$
4: $a_c \leftarrow a$
5: **while** $n_j > 3$ **do**
6: $\quad j_3 \leftarrow_R J$ // Escoger aleatoriamente un elemento de J
7: $\quad \{j_0, j_1, j_2\} \leftarrow_R J$
8: $\quad f_c \leftarrow CF$ de cuatro 3-Cláusulas como se menciona en la Sección 2.1.3 y en el ejemplo mostrado, involucrando las variables $X_{j_0}, X_{j_1}, X_{j_2}, X_{j_3}$ de manera que $Q^{n_j}|_{x_{j_3} \neq \delta_3} \subset \text{Nul}(f_c)$
9: $\quad f_a \leftarrow f_a \cup f_c$
10: $\quad$ Eliminar de a_c su j_3-ésima entrada
11: $\quad J \leftarrow J - \{j_3\}$
12: **end while**
13: $\{j_0, j_1, j_2\} \leftarrow J$
14: $f_c \leftarrow f_c \cup f_{3,a_c}(X_{j_0}, X_{j_1}, X_{j_2})$ // como se dió en la Ecuación 2.2
15: **return** f_c

2.2 Polinomios cuadráticos en varias variables

Revisaremos ahora cómo construir un polinomio cuadrático en varias variables así como el número de términos[1] que como máximo puede tener dicho polinomio. Esto está relacionado directamente con el tamaño de las claves públicas que se pueden generar en un sistema criptográfico basado en polinomios cuadráticos, ya que mientras

[1]Existe una dualidad en el uso las palabras término y monomio. En álgebra un término es un producto de la forma $x_1^{\beta_1} \cdots x_n^{\beta_n}$ de un polinomio. La palabra término se usa también comúnmente para nombrar un sumando de un polinomio que incluye su coeficiente. Sin embargo, esto debería llamarse propiamente monomio. Pero para estar de acuerdo con los libros que tratan bases de Gröbner, en este trabajo usaremos término como lo que normalmente designa a un monomio, i.e. un coeficiente que multiplica un producto de potencias: $ax_1^{\beta_1} \cdots x_n^{\beta_n}$.

más variables y de mayor grado es el conjunto de polinomios, más grande también será dicha clave.

Por otra parte, demostraremos que encontrar un conjunto de soluciones a un sistema de polinomios cuadráticos, es un problema NP-Completo. Esto implica que la resolución de este tipo de problemas es generalmente difícil, lo que no necesariamente significa que sea imposible resolver.

2.2.1 Construcción de ecuaciones cuadráticas de varias variables

Definimos a continuación lo que entendemos por sistema de polinomios en varias variables:
Sea $n \in \mathbb{N}$ el número de variables que pueden aparecer en los polinomios del sistema, $m \in \mathbb{N}$ el número de polinomios y $d \in \mathbb{N}$ el grado de este sistema. Las variables $X_1, \ldots, X_n$ tomarán valores sobre un campo finito[2] $\mathbb{F}$ y la variable X_0 por convención se toma como uno. Además, para n y d dados, se define:

$$\nu_n^d := \begin{cases} \{0\} & \text{para d} = 0 \quad (2.3a) \\ \{u \in \{0,\ldots,n\}^d : i \leq j \Rightarrow u_i \leq u_j\} & \text{en otro caso} \quad (2.3b) \end{cases}$$

Y los componentes del vector u se denotan como $u_1, \ldots, u_d \in \{0, \ldots, n\}$. Por otra parte, si $\mathcal{P}$ es un sistema de m polinomios, en n variables, con máximo grado d cada uno, entonces se tiene que $\mathcal{P} = \{p_1, \ldots, p_m\}$ y cada p_i tiene la forma:

$$p_i(X_1, \ldots, X_n) := \sum_{u \in \nu_n^d} \gamma_{i,u} \prod_{j=1}^{d} X_{u_j} \; para \; 1 \leq i \leq m \quad (2.4)$$

Con $\gamma_{i,u} \in \mathbb{F}$ ($\mathbb{F}$, algún campo). Si $d = 2$, se tiene el caso de *ecuaciones cuadráticas de varias variables* y de la Ecuación (2.3b) vemos que u será un vector perteneciente al conjunto ν_n^d, donde cada vector estará formado por *dos* números (índices) entre 0 y n. Por ejemplo, u podría tomar los valores $(0,0), (0,1), (1,1), (1,2)$, etc. Esto nos generará tres casos:

[2]Un anillo $(R, +, \times)$, consiste de un conjunto R con dos operaciones binarias denotadas arbitrariamente como suma $+$ y multiplicación $\times$ en R y que satisfacen los siguientes axiomas:

I. $(R, +)$ es un grupo abeliano con identidad denotada como 0

II. La operación $\times$ es asociativa: $a \times (b \times c) = (a \times b) \times c$ para toda $a, b, c \in R$

III. Hay una identidad multiplicativa denotada como 1, con $1 \neq 0$, tal que $1 \times a = a \times 1 = a$ para toda $a \in R$

IV. La operación $\times$ es distributiva sobre $+$: $a \times (b+c) = (a \times b) + (a \times c)$ y $(b+c) \times a = (b \times a) + (c \times a)$ para toda $a, b, c \in R$

Un anillo es un *anillo conmutativo* si $a \times b = b \times a$ para toda $a, b \in R$.
Un campo es un anillo conmutativo y un campo finito es un campo que contiene un número finito de elementos.

1. Cuando u toma el valor $(0,0)$, entonces el producto que se tiene en la Ecuación (2.4) es $\prod_{j=1}^{2} X_{u_j} = X_0 \cdot X_0 = 1 \cdot 1 = 1$, dado que X_0 como se mencionó antes, se toma como 1 y en este caso se obtiene solamente la constante 1.

2. Si u toma valores como $(0,1),(0,2),\ldots$, es decir sólo el primer índice toma el valor de cero, entonces el producto de la Ecuación (2.4) para el caso $(0,1)$ es $\prod_{j=1}^{2} X_{u_j} = X_0 \cdot X_1 = 1 \cdot X_1 = X_1$, esto es, el monomio que se obtiene es en una sola variable.

3. Cuando u no toma a ninguno de sus índices como cero (por ejemplo $u = (1,2)$), entonces el producto de la Ecuación (2.4) es $\prod_{j=1}^{2} X_{u_j} = X_1 X_2$, es decir, se tiene un monomio de segundo orden porque se multiplican dos variables X_i diferentes.

Considerando los puntos anteriores vemos entonces, que en el caso de *ecuaciones cuadráticas de varias variables* tendremos la suma de términos cuadráticos (producto de dos variables X), con términos lineales (una sola variable X) y una constante. Por otra parte, el coeficiente $\gamma_{i,u}$ de la Ecuación (2.4) podemos ahora dividirlo en tres, uno para cada caso de los mencionados antes; sean estos: γ, β y α. Entonces, los polinomios tomarán la siguiente forma:

$$p_i(X_1,\ldots,X_n) := \sum_{1\le j\le k\le n} \gamma_{i,j,k} X_j X_k + \sum_{j=1}^{n} \beta_{i,j} X_j + \alpha_i \tag{2.5}$$

Donde $1 \le i \le m$; $\gamma_{i,j,k}, \beta_{i,j}, \alpha_i \in \mathbb{F}$. Generalmente a $\gamma_{i,j,k}$ se le conoce como coeficiente cuadrático, a $\beta_{i,j}$ como coeficiente lineal y a α_i como el coeficiente constante. En el caso de $\mathbb{F}_2$, $j < k$ ya que $X_i^2 = X_i$. Entonces, el número máximo de términos (τ) que se tendrán en la Ecuación 2.5 para $\mathbb{F}_2$ es proporcional al binomial $\binom{n}{2}$ y para cualquier otro campo (donde $X_i^2 \neq X_i$) será proporcional a $\sum_{i=1}^{n} i$. De manera exacta (considerando el término constante, las variables solas y los términos cuadráticos):

$$\tau(n) = \begin{cases} 1 + n + \dfrac{n(n-1)}{2} = 1 + \dfrac{n(n+1)}{2} & \text{en } \mathbb{F}_2 \quad (2.6a) \\ 1 + n + \dfrac{n(n+1)}{2} = 1 + \dfrac{n(n+3)}{2} & \text{en otro caso} \quad (2.6b) \end{cases}$$

Entonces, un sistema de polinomios en general, tendrá un tamaño $O(mn^d)$. Si este sistema de polinomios, representara una clave pública, vemos que ésta crece con el número de variables y el grado de los polinomios, por lo que es deseable tener el grado d lo más pequeño posible, como en este caso, grado dos para polinomios cuadráticos. Por otra parte, el problema de resolver un sistema de ecuaciones cuadráticas es NP-Completo y difícil en general. En la Sección 2.2.2 demostramos que un sistema de ecuaciones cuadráticas en varias variables es un problema NP-Completo.

2.2.2 Ecuaciones cuadráticas: NP-Completo

En (Garey and Johnson, 1990, p. 251), se menciona que el problema de resolver un sistema de ecuaciones cuadráticas en un campo $\mathbb{F}_2$ es NP-Completo. A continuación,

se da la prueba de esto:

INSTANCIA: Un conjunto de polinomios $p_i(X_1, \ldots, X_n)$, $1 \leq i \leq m$, sobre $\mathbb{F}_2$, donde cada polinomio es una suma de términos y cada término es la constante 1 o un producto de variables diferentes X_i.

PREGUNTA: ¿Existen $u_1, \ldots, u_n \in \{0, 1\}$ tal que para $1 \leq i \leq m$, $p_i(u_1, \ldots, u_n) = 0$?

Consideremos una instancia del problema 3-SAT, dado por un conjunto finito $U = \{u_1, \ldots, u_n\}$ de variables booleanas y una colección $C = \{c_1, \ldots, c_m\}$ de cláusulas. Por definición, cada cláusula es una disyunción de a lo más tres literales en U, donde una literal es alguna u o $\bar{u}$ con $u \in U$. Cada variable booleana será evaluada como un elemento de $\mathbb{F}_2$ y se define una correspondencia entre literales y las expresiones aritméticas en $\mathbb{F}_2$. Así, realizamos las siguientes consideraciones:

- Si a $u \in U$, le corresponde $x \in \mathbb{F}_2$, entonces a $\bar{u}$ le corresponde $1 - x$.
- Si a $u, v \in U$ le corresponden $x_1, x_2 \in \mathbb{F}_2$, entonces a $(u \vee v)$ le corresponde $(x_1 + x_2 + x_1x_2)$ en $\mathbb{F}_2$.
- Así mismo, si a $u, v, w \in U$ le corresponden $x_1, x_2, x_3 \in \mathbb{F}_2$ entonces a $(u \vee v \vee w)$ le corresponde $((x_1 + x_2 + x_1x_2) + x_3 + (x_1 + x_2 + x_1x_2)x_3)$ o bien $(x_1 + x_2 + x_1x_2 + x_3 + x_1x_3 + x_2x_3 + x_1x_2x_3)$

Entonces, encontrar una asignación verdadera en U, que satisfaga todas las cláusulas en C, es equivalente a resolver un sistema cúbico en m ecuaciones y en n variables. Por otra parte, cada ecuación de este sistema contiene a lo más tres variables x_i y de estas se forman $\binom{n}{2} = \frac{n(n-1)}{2}$ pares de variables que se llamarán $y_{ij} = x_ix_j$ con $i < j$. Al sustituir estas nuevas variables y_{ij} en el sistema de ecuaciones, se forma ahora un sistema cuadrático de $m + \frac{n(n-1)}{2}$ ecuaciones en $n + \frac{n(n-1)}{2} = \frac{n(n+1)}{2}$ variables en $\mathbb{F}_2$.

Con esto, se ve que resolver una instancia aleatoriamente seleccionada del problema 3-SAT, mismo que es NP-Completo, se puede reducir en tiempo polinomial al problema de resolver un sistema aleatoriamente seleccionado de ecuaciones cuadráticas en $\mathbb{F}_2$. Por lo tanto, un sistema cuadrático de ecuaciones es también NP-Completo.

2.3 Bases de Gröbner

Las técnicas de bases de Gröbner proporcionan soluciones algorítmicas a una gran variedad de problemas en álgebra conmutativa y en geometría algebraica. Una de estas aplicaciones es la solución de ecuaciones simultaneas no lineales y de ahí su importancia en el tema general que tratamos, ya que solucionar el conjunto de ecuaciones, que se presentan como una clave pública, podría ser resuelto por esta metodología.

El originador del concepto de las bases de Gröbner es Bruno Buchberger, quien en 1965 presentó un algoritmo para el cálculo de estas. En esta sección, presentamos las definiciones necesarias que están involucradas en este tema, orden que considera guardan los términos en un polinomio, un algoritmo para divisón de polinomios y

con estos elementos, el algoritmo de Buchberger. Finalmente, mencionaremos algunos métodos que han mejorado al algoritmo original.

2.3.1 Definiciones

Consideraremos polinomios $f(x_1, \ldots, x_n)$ en n variables y con coeficientes en un campo $\mathbb{K}$. Entonces $\mathbb{K}[x_1, \ldots, x_n]$ denota un anillo de polinomios. Los términos de un polinomio toman la forma $ax_1^{\beta_1} \cdots x_n^{\beta_n}$ donde $a \in \mathbb{K}$ y $\beta_i \in \mathbb{N}$[3], $i = 1, \ldots, n$. Llamamos a $x_1^{\beta_1} \cdots x_n^{\beta_n}$ producto potencia. Al anillo de polinomios $\mathbb{K}[x_1, \ldots, x_n]$ se le puede ver también como un espacio $\mathbb{K}$-vectorial con base en el conjunto de todos los productos potencia:

$$\mathbb{T}^n = \{x_1^{\beta_1} \cdots x_n^{\beta_n} | \beta_i \in \mathbb{N}, i = 1, \ldots, n\} \tag{2.7}$$

Un polinomio, además de verse como un elemento del anillo de polinomios $\mathbb{K}[x_1, \ldots, x_n]$, se puede ver también como una función $\mathbb{K}^n \to \mathbb{K}$ cuando

$$(a_1, \ldots, a_n) \mapsto f(a_1, \ldots, a_n), \forall (a_1 \ldots, a_n) \in \mathbb{K}^n.$$

La *variedad* definida por f, es:

$$V(f) = \{(a_1, \ldots, a_n) \in \mathbb{K}^n | f(a_1, \ldots, a_n) = 0\} \subseteq \mathbb{K}^n$$

Más generalmente, se puede definir una *variedad* para varias funciones:

$$V(f_1, \ldots, f_s) = \{(a_1, \ldots, a_n) \in \mathbb{K}^n | f_i(a_1, \ldots, a_n) = 0, i = 1, \ldots, n\}.$$

Para obtener información algebráica y geométrica del espacio de solución completo de una variedad, lo haremos por medio del ideal generado por los polinomios $f_1, \ldots, f_s$ y que denotaremos como $I = \langle f_1, \ldots, f_s \rangle$:

$$\langle f_1, \ldots, f_s \rangle = \left\{ \sum_{i=1}^{s} u_i f_i | u_i \in \mathbb{K}[x_1, \ldots, x_n], i = 1, \ldots, s \right\}$$

I es un ideal, ya que si $f, g \in I$ también lo estará $f + g$ y si $f \in I$ y h es cualquier polinomio en $\mathbb{K}[x_1, \ldots, x_n]$ entonces $hf \in I$.

Se puede apreciar que $V(I)$ (i.e. la solución al sistema de polinomios infinito $f = 0, f \in I$), será también una solución a $V(f_1, \ldots, f_s)$. Vemos de lo antes expuesto, que un ideal puede tener muchos conjuntos generadores cada uno con diferente número de elementos. Entonces, si $I = \langle f_1, \ldots, f_s \rangle = \langle f_1', \ldots, f_t' \rangle$ y $V(f_1, \ldots, f_s) = V(I) = V(f_1', \ldots, f_t')$. Podemos concluir entonces que la solución de $f_1 = 0, \ldots, f_s = 0$ es la misma que de $f_1' = 0, \ldots, f_t' = 0$ y por lo tanto una variedad está determinada por un ideal y no por un conjunto particular de ecuaciones. Encontrar un mejor conjunto generador del ideal $I = \langle f_1, \ldots, f_s \rangle$ producirá una mejor representación de la variedad $V(f_1, \ldots, f_s)$. Este mejor conjunto generador para I será precisamente la base de Gröbner de I.

[3] Consideramos que $\mathbb{N} = \{0, 1, 2, \ldots\}$

Teorema 2.3.1 *Teorema de Hilbert de las bases.*[4] *Cada ideal* $I \in \mathbb{K}[x_1, \ldots, x_n]$ *tiene un conjunto generador finito. Esto es,* $I = \langle f_1, \ldots, f_s \rangle$ *para algunas* $f_1, \ldots, f_s \in I$

Al proceso por el que un polinomio f_2 se sustituye por otro f_3, usando f_1 se llama reducción de f_2 por f_1 y lo escribimos así:

$$f_2 \xrightarrow{f_1} f_3$$

f_3 es el residuo de la división $\frac{f_2}{f_1}$. Si tenemos varios pasos de reducción consecutivos:

$$f \xrightarrow{g} h \xrightarrow{g} r$$

Lo denotaremos como:

$$f \xrightarrow{g}_{+} r$$

2.3.2 Orden de los términos

Cuando se tienen varios términos de un polinomio en varias variables es necesario especificar un orden para operar con dichos polinomios. No importa qué orden se tome, siempre y cuando este sea consistente en todos los polinomios. En algunos casos, de la ecuación del producto de potencias 2.7, se denota $x_1^{\beta_1} \cdots x_n^{\beta_n}$ como x^β donde $\beta = (\beta_1, \ldots, \beta_n) \in \mathbb{N}^n$. El orden que se guarda en los términos de un polinomio, debe ser de *orden total*, es decir, dados $x^\alpha, x^\beta \in \mathbb{T}^n$ entonces, exactamente una de las tres relaciones siguientes se debe cumplir:

$$x^\alpha < x^\beta, x^\alpha = x^\beta, x^\alpha > x^\beta$$

El ordenamiento es importante, ya que para que una reducción (como se mencionó al final de la sección anterior) pare después de un número finito de pasos, no debemos tener una cadena descendente infinita de pasos $x^{\alpha_1} > x^{\alpha_2} > \cdots \in \mathbb{T}$. Aplica entonces la siguiente definición:

Definición 2.3.1 *Un orden de términos en* $\mathbb{T}^n$ *es un orden total* $<$ *en* $\mathbb{T}^n$ *que satisface las siguientes dos condiciones:*

I. $1 < x^\beta$ *para toda* $x^\beta \in \mathbb{T}^n, x^\beta \neq 1$

II. *Si* $x^\alpha < x^\beta$ *entonces* $x^\alpha x^\gamma < x^\beta x^\gamma, \forall x^\gamma \in \mathbb{T}^n$

Consideremos las siguientes tres formas de orden de términos:

Definición 2.3.2 *El orden lexicográfico en* $\mathbb{T}^n$ *con* $x_1 > x_2 > \cdots x_n$ *para* $\alpha = (\alpha_1, \ldots, \alpha_n)$, $\beta = (\beta_1, \ldots, \beta_n)$ *es:* $\alpha >_{lex} \beta$ *si, en la diferencia de vectores* $\alpha - \beta \in \mathbb{N}^n$ *la entrada extrema a la izquierda diferente de cero, es positiva. Lo escribimos como* $x^\alpha >_{lex} x^\beta$ *si* $\alpha >_{lex} \beta$

[4] Para una demostración de este teorema, ver por ejemplo (Adams and Loustaunau, 1994, p. 5) o (Cox et al., 2015, p. 76)

Definición 2.3.3 *El orden lexicográfico determinado por grado (degree lexicographical) en $\mathbb{T}^n$ con $x_1 > x_2 > \cdots x_n$ para $\alpha = (\alpha_1, \ldots, \alpha_n)$, $\beta = (\beta_1, \ldots, \beta_n)$ es: $\alpha >_{deglex} \beta$ si*

$$|\alpha| = \sum_{i=1}^{n} \alpha_i > |\beta| = \sum_{i=1}^{n} \beta_i \text{ o } |\alpha| = |\beta| \text{ y } \alpha >_{lex} \beta$$

Definición 2.3.4 *El orden lexicográfico inverso determinado por grado (degree reverse lexicographical) en $\mathbb{T}^n$ con $x_1 > x_2 > \cdots x_n$ para $\alpha = (\alpha_1, \ldots, \alpha_n)$, $\beta = (\beta_1, \ldots, \beta_n)$ es: $\alpha >_{degrevlex} \beta$ si*

$$|\alpha| = \sum_{i=1}^{n} \alpha_i > |\beta| = \sum_{i=1}^{n} \beta_i \text{ o } |\alpha| = |\beta| \text{ y la entrada extrema derecha diferente de cero de } \alpha - \beta \in \mathbb{N}^n \text{ es negativa.}$$

Definición 2.3.5 *Establecemos la siguiente notación: habiendo escogido un orden de términos en $\mathbb{K}[x_1, \ldots, x_n]$, entonces, para todo polinomio $f \in \mathbb{K}[x_1, \ldots, x_n]$ con $f \neq 0$ podemos escribir*

$$f = a_1 x^{\alpha_1} + \cdots + a_r x^{\alpha_r},$$

donde $0 \neq a_i \in \mathbb{K}$, $x^{\alpha_i} \in \mathbb{T}^n$, y $x^{\alpha_1} > x^{\alpha_2} > \cdots > x^{\alpha_r}$. Definimos entonces:

- $\mathrm{lp}(f) = x^{\alpha_1}$ *el producto potencia principal de f*
- $\mathrm{lc}(f) = a_1$ *el coeficiente principal de f*
- $\mathrm{lt}(f) = a_1 x^{\alpha_1}$ *el término principal de f*

2.3.3 Algoritmo de división

Estudiaremos ahora un algoritmo para división en $\mathbb{K}[x_1, \ldots, x_n]$. Cuando dividimos f entre $f_1, \ldots, f_s$, buscamos cancelar términos de f usando los términos principales de las f_i's de manera que los nuevos términos introducidos, sean menores a los términos cancelados. Luego se continúa este proceso hasta que no sea posible ninguna reducción adicional.

Definición 2.3.6 *Para el caso especial de f entre g cuando $f, g, h \in \mathbb{K}[x_1, \ldots, x_n]$ y $g \neq 0$, decimos que f se reduce a h módulo g en un paso y lo escribimos como:*

$$f \xrightarrow{g} h,$$

si y sólo si $\mathrm{lp}(g)$ divide un término diferente de cero X que aparece en f y

$$h = f - \frac{X}{\mathrm{lt}(g)} g.$$

Con esto, eliminamos completamente el término X de f y los sustituimos estrictamente por términos menores que X. Este proceso puede continuar y eliminar entonces de f todos los términos que sean divisibles por $\mathrm{lt}(g)$.

Definición 2.3.7 *Sean* f, h *y* $f_1, \ldots, f_s \in \mathbb{K}[x_1, \ldots, x_n]$ *polinomios con* $f_i \neq 0$ $(1 \leq i \leq s)$ *y sea* $F = \{f_1, \ldots, f_s\}$. *Entonces, decimos que* f *se reduce a* h *modulo* F *denotado como:*

$$f \xrightarrow{F}_{+} h,$$

si y sólo si existe una sucesión de índices $i_1, \ldots, i_t \in \{1, \ldots, s\}$ *y una sucesión de polinomios* $h_1, \ldots, h_{t-1} \in \mathbb{K}[x_1, \ldots, x_n]$ *tal que*

$$f \xrightarrow{f_{i_1}} h_1 \xrightarrow{f_{i_2}} h_2 \xrightarrow{f_{i_3}} \cdots \xrightarrow{f_{i_{t-1}}} h_{t-1} \xrightarrow{f_{i_t}} h$$

Definición 2.3.8 *Un polinomio* r *se llama reducido con respecto a un conjunto de polinomios diferentes de cero* $F = \{f_1, \ldots, f_s\}$ *si* $r = 0$ *o ningún producto potencia que aparezca en* r *es divisible por cualquiera de los términos* $\mathrm{lp}(f_i)$, $i = 1, \ldots, s$. *En otras palabras,* r *no puede ser reducido modulo* F.

Definición 2.3.9 *Si* $f \xrightarrow{F}_{+} r$ *y* r *está reducido respecto a* F *entonces llamamos a* r *un residuo para* f *con respecto a* F.

El proceso de reducción nos permite definir un algoritmo de división. Entonces, dados $f, f_1, \ldots, f_s \in \mathbb{K}[x_1, \ldots, x_n]$ con $f_i \neq 0, (i = 1, \ldots, s)$ este algoritmo regresa los cocientes $u_1, \ldots, u_s \in \mathbb{K}[x_1, \ldots, x_n]$ tal que

$$f = u_1 f_1 + \cdots + u_s f_s + r.$$

Así, el algoritmo de división para varias variables lo encontramos en el Algoritmo 2.

Algoritmo 2 Algoritmo de división para varias variables

Entrada: $f, f_1, \ldots, f_s \in \mathbb{K}[x_1, \ldots, x_n]$ con $f_i \neq 0, (1 \leq i \leq s)$
Salida: $u_1, \ldots, u_s, r$ tal que $f = u_1 f_1 + \cdots + u_s f_s + r$ y r está reducido respecto a $f_1, \ldots, f_s$ y $\max(\mathrm{lp}(u_1)\mathrm{lp}(f_1), \ldots, \mathrm{lp}(u_s)\mathrm{lp}(f_s), \mathrm{lp}(r)) = \mathrm{lp}(f)$

1: $u_1 \leftarrow 0, \ldots, u_s \leftarrow 0$
2: $r \leftarrow 0$
3: $h \leftarrow f$
4: **while** $h \neq 0$ **do**
5: **if** Existe un i tal que $\mathrm{lp}(f_i)$ divide $\mathrm{lp}(h)$ **then**
6: Escoger el i menor tal que $\mathrm{lp}(f_i)$ divida $\mathrm{lp}(h)$
7: $u_i \leftarrow u_i + \dfrac{\mathrm{lt}(h)}{\mathrm{lt}(f_i)}$
8: $h \leftarrow h - \dfrac{\mathrm{lt}(h)}{\mathrm{lt}(f_i)} f_i$
9: **else**
10: $r \leftarrow r + \mathrm{lt}(h)$
11: $h \leftarrow h - \mathrm{lt}(h)$
12: **end if**
13: **end while**
14: **return** $u_1, \ldots, u_s, r$

Este algoritmo permite expresar a $f = u_1 f_1 + \cdots + u_s f_s + r$, de donde $f - r \in \langle f_1, \ldots, f_s \rangle$. Y si $r = 0$ entonces $f \in \langle f_1, \ldots, f_s \rangle$, y tenemos la solución al llamado problema de pertenencia a un ideal. La condición suficiente de esta pertenencia a un ideal es entonces $r = 0$. Lo contrario no es necesariamente cierto:

La línea 6 del Algoritmo 2 implica que el conjunto de funciones $\{f_i, \ldots, f_s\}$ tiene un orden para poder elegir el i menor tal que $\mathrm{lp}(f_i)$ divida $\mathrm{lp}(h)$. Esto es importante, ya que para la misma serie de funciones f_i el orden en que se van tomando en el algoritmo puede producir resultados diferentes. El ejemplo 1.5.10 de (Adams and Loustaunau, 1994, Cap. 1, Sec. 1.5, p. 31) muestra como si $f = y^2x - x$ y $f_1 = yx - y$, $f_2 = y^2 - x$, si se toma primero a f_1 y luego a f_2 para reducir f, se encuentra $f = yf_1 + f_2$ y como en este caso $r = 0$, $f \in I = \langle f_1, f_2 \rangle$. Sin embargo si usamos primero f_2 para reducir f, se tiene que $f = xf_2 + (x^2 - x)$ obteniéndose un residuo diferente de cero aunque f pertenece al ideal I. Para resolver este problema, es necesario tener un mejor conjunto generador del ideal, que para el caso de polinomios en varias variables, esto nos lo proporcionarán las bases de Gröbner.

2.3.4 Algoritmo de Buchberger

Antes de mostrar el algoritmo que sirve para calcular la base de Gröbner para un ideal I, definiremos qué es tal base de Gröbner y qué son los polinomios S, mismos que se usan en el algoritmo de Buchberger.

Definición 2.3.10 *Un conjunto de polinomios diferentes de cero* $G = \{g_1, \ldots, g_t\}$ *contenidos en un ideal* I*, se conocen como base de Gröbner para* I *si y sólo si para toda* $f \in I$ *tal que* $f \neq 0$ *existe un índice* $i \in \{1, \ldots, t\}$ *tal que* $\mathrm{lp}(g_i)$ *divide a* $\mathrm{lp}(f)$*.*

Es decir, si G es una base de Gröbner para I, entonces no hay polinomios diferentes de cero en I reducidos respecto a G. Por otra parte, para un subconjunto S de $\mathbb{K}[x_1, \ldots, x_n]$ definimos el ideal de términos principales de S como $\mathrm{Lt}(S) = \langle \mathrm{lt}(s) | s \in S \rangle$. Con esto, se tiene lo siguiente:

Teorema 2.3.2 *Sea* I *un ideal diferente de cero de* $\mathbb{K}[x_1, \ldots, x_n]$*. Las siguientes afirmaciones son equivalentes para un conjunto de polinomios diferentes de cero* $G = \{g_1, \ldots, g_t\} \subseteq I$[5]

I. G *es una base de Gröbner para* I

II. $f \in I$ *si y sólo si* $f \xrightarrow{G}_+ 0$

III. $f \in I$ *si y sólo si* $f = \sum_{i=1}^{t} h_i g_i$ *con* $\mathrm{lp}(f) = \max_{1 \leq i \leq t}(\mathrm{lp}(h_i)\mathrm{lp}(g_i))$

IV. $\mathrm{Lt}(G) = \mathrm{Lt}(I)$

Corolario 2.3.1 *Si* $G = \{g_1, \ldots, g_t\}$ *es una base de Gröbner para un ideal* I*, entonces* $I = \langle g_1, \ldots, g_t \rangle$

Lemma 2.3.1 *Sea* I *un ideal generado por un conjunto* S *de polinomios diferentes de cero, y sea* $f \in \mathbb{K}[x_1, \ldots, x_n]$*. Entonces,* $f \in I$ *si y sólo si para cada término* X *que aparece en* f *existe un* $Y \in S$ *de forma tal que ese* Y *divide a* X*. Es más, existe un subconjunto finito* S_0 *de* S *tal que* $I = \langle S_0 \rangle$*.*

Por otra parte, si por cada término X *que aparece en* f *existe un término* $Y \in S$ *tal que* Y *divida a* X *entonces tales* X*'s están en* $I = \langle S \rangle$ *y por lo tanto* f *está en* I*.*

Corolario 2.3.2 *Todo ideal* I *diferente de cero de* $\mathbb{K}[x_1, \ldots, x_n]$ *tiene una base de Gröbner.*

Definición 2.3.11 *Un subconjunto* $G = \{g_1, \ldots, g_t\}$ *de* $\mathbb{K}[x_1, \ldots, x_n]$ *es una base de Gröbner si y sólo si es una base de Gröbner para el ideal* $\langle G \rangle$ *que genera.*

Teorema 2.3.3 *Sea* $G = \{g_1, \ldots, g_t\}$ *un conjunto de polinomios diferentes de cero en* $\mathbb{K}[x_1, \ldots, x_n]$*. Entonces,* G *es una base de Gröbner si y sólo si para toda* $f \in \mathbb{K}[x_1, \ldots, x_n]$ *el residuo de la división de* f *por* G *es único.*

De acuerdo a la Definición 2.3.10, $F = \{f_1, \ldots, f_s\}$ es una base de Gröbner si y sólo si para toda $f \in I$ existe un $i \in \{1, \ldots, s\}$ tal que $\mathrm{lp}(f_i)$ divide a $\mathrm{lp}(f)$. Entonces, surge un problema cuando se tienen elementos en I cuyos productos potencia principales, no son divisibles por alguno de los productos potencia principales de f_i. Como $f \in I$ entonces $f = \sum_{i=1}^{s} h_i f_i$ para alguna $h_i \in \mathbb{K}[x_1, \ldots, x_n]$. El problema surge entonces cuando el más grande de los productos potencia principal, $\mathrm{lp}(h_i f_i) = \mathrm{lp}(h_i)\mathrm{lp}(f_i)$ se cancela. Esto puede ocurrir fácilmente en el caso de polinomios-S definidos como:

[5] Ver demostración en (Adams and Loustaunau, 1994, pp. 32-33)

Definición 2.3.12 *Sean* $0 \neq f, g \in \mathbb{K}[x_1, \ldots, x_n]$. *Sea* $L = \text{mcm}(\text{lp}(f), \text{lp}(g))$[6]. *El polinomio*

$$S(f,g) = \frac{L}{\text{lt}(f)} f - \frac{L}{\text{lt}(g)} g$$

se le llama polinomio-S de f *y* g.

Teorema 2.3.4 (Buchberger) *Sea* $G = \{g_1, \ldots, g_t\}$ *un conjunto de polinomios diferentes de cero, en* $\mathbb{K}[x_1, \ldots, x_n]$. G *es una base de Gröbner para el ideal* $I = \langle g_1, \ldots, g_t \rangle$ *si y sólo si para toda* $i \neq j$, $S(g_i, g_j) \xrightarrow{G}_+ 0$[7].

Corolario 2.3.3 *Sea* $G = \{g_1, \ldots, g_t\}$ *con* $g_i \neq 0$ $(1 \leq i \leq t)$. *Entonces* G *es una base de Gröbner si y sólo si para toda* $i \neq j$ $(1 \neq i, j \neq t)$, *tenemos*

$$S(g_i, g_j) = \sum_{\nu=1}^{t} h_{ij\nu} g_\nu, \text{ donde } \text{lp}(S(g_i, g_j)) = \max_{1 \leq \nu \leq t}(\text{lp}(h_{ij\nu}), \text{lp}(g_\nu))$$

.

Así, el Teorema 2.3.4 nos da una estrategia para el cálculo de las bases de Gröbner: reducir los polinomios-S y si un residuo es diferente de cero, agregar este residuo a la lista de polinomios en el conjunto generador. Repetir esto, hasta que haya suficientes polinomios para hacer que todos los polinomios-S se reduzcan a cero.

Para resumir los conceptos anteriores, veamos un ejemplo:
Sea $f_1 = yx - x, f_2 = x^2 - y \in \mathbb{Q}[x, y]$ con el orden de términos *deglex* y $x < y$. Sea $F = \{f_1, f_2\}$. Entonces:

1. El mínimo común múltiplo $L = \text{mcm}(\text{lp}(f_1), \text{lp}(f_2)) = \text{mcm}(yx, x^2) = yx^2$
2. El polinomio-S es $S(f_1, f_2) = \frac{L}{\text{lt}(f_1)} f_1 - \frac{L}{\text{lt}(f_2)} f_2 = \frac{yx^2}{yx}(yx - x) - \frac{yx^2}{x^2}(x^2 - y)$

$$S(f_1, f_2) = yx^2 - x^2 - yx^2 + y^2 = y^2 - x^2$$

3. Reduzcamos $S(f_1, f_2) \xrightarrow{F} y^2 - y$, esto al dividir entre f_2.
4. Sea $f_3 = y^2 - y$, que está reducida respecto a F.
5. Agregamos f_3 a F y ponemos $F' = \{f_1, f_2, f_3\}$
6. Entonces $S(f_1, f_2) \xrightarrow{F'} 0$.
7. Por otra parte, $L' = \text{mcm}(\text{lp}(f_1), \text{lp}(f_3)) = \text{mcm}(yx, y^2) = y^2 x$

[6] El mínimo común múltiplo mcm (*least common multiple* de sus siglas en inglés) de dos productos potencia X, Y es el producto potencia L tal que $X \mid L$, $Y \mid L$ y si Z es otro producto potencia tal que $X \mid Z$ y $Y \mid Z$ entonces $L \mid Z$.

[7] Ver demostración en (Adams and Loustaunau, 1994, pp. 40-42)

8. Entonces $S(f_1, f_3) = \frac{y^2x}{xy}(xy - x) - \frac{y^2x}{y^2}(y^2 - y) = y^2x - yx - y^2x + yx = 0$

9. $L'' = \text{mcm}(\text{lp}(f_2), \text{lp}(f_3)) = \text{mcm}(x^2, y^2) = y^2x^2$

10. Y $S(f_2, f_3) = \frac{y^2x^2}{x^2}(x^2 - y) - \frac{y^2x^2}{y^2}(y^2 - y) = y^2x^2 - y^3 - y^2x^2 + yx^2 = -y^3 + yx^2$

11. Finalmente $S(f_2, f_3) \xrightarrow{F'} -yx^2 + y^2 \xrightarrow{F'} -y^2 + x^2 \xrightarrow{F'} x^2 - y \xrightarrow{F'} 0$ (usando primero f_3, luego f_1, nuevamente f_3 y terminando con f_2)

Debido a lo anterior, concluimos que $\{f_1, f_2, f_3\}$ es una base de Gröbner.

Teorema 2.3.5 [8] *Dado $F = \{f_1, \ldots, f_s\}$ con $f_i \neq 0$ $(1 \leq i \leq s)$, el Algoritmo de Buchberger 3 producirá una base de Gröbner para el ideal $I = \langle f_1, \ldots, f_s \rangle$*

Algoritmo 3 Algoritmo de Buchberger para el cálculo de bases de Gröbner

Entrada: $F = \{f_1, \ldots, f_s\} \subseteq \mathbb{K}[x_1, \ldots, x_n]$ con $f_i \neq 0, (1 \leq i \leq s)$
Salida: $G = \{g_1, \ldots, g_t\}$, una base de Gröbner para $\langle f_1, \ldots, f_s \rangle$

1: $G \leftarrow F$
2: $\mathcal{G} \leftarrow \{\{f_i, f_j\} | f_i \neq f_j \in G\}$
3: **while** $\mathcal{G} \neq 0$ **do**
4: Escoger cualquier par de funciones $\{f, g\} \in \mathcal{G}$
5: $\mathcal{G} \leftarrow \mathcal{G} - \{\{f, g\}\}$
6: $S(f, g) \xrightarrow{G}_+ h$, donde h está reducida respecto a G
7: **if** $h \neq 0$ **then**
8: $\mathcal{G} \leftarrow \mathcal{G} \cup \{\{u, h\} \forall u \in G\}$
9: $G \leftarrow G \cup \{h\}$
10: **end if**
11: **end while**
12: **return** G

2.3.5 Bases de Gröbner reducidas

El orden en que los polinomios $F = \{f_1, \ldots, f_s\} \subseteq \mathbb{K}[x_1, \ldots, x_n]$ se dan en el Algoritmo 3 así como las decisiones que se toman en el punto 4 del mismo algoritmo, pueden ocasionar que para el mismo conjunto de funciones, se pueda terminar con una base de Gröbner diferente. Tenemos entonces la siguiente

Definición 2.3.13 *Una base de Gröbner $G = \{g_1, \ldots, g_t\}$ se conoce como mínima si para toda i, $\text{lc}(g_i) = 1$ y para toda $i \neq j$ $\text{lp}(g_i)$ no divide a $\text{lp}(g_j)$.*

[8] Ver demostración en (Adams and Loustaunau, 1994, pp. 42-43)

Lemma 2.3.2 *Sea* $G = \{g_1, \ldots, g_t\}$ *una base de Gröbner para un ideal* I*. Si* $\mathrm{lp}(g_2)$ *divide a* $\mathrm{lp}(g_1)$*, entonces* $\{g_2, \ldots, g_t\}$ *también es una base de Gröbner para* I*.*

Con base en este lemma, se puede ahora generar una base de Gröbner mínima con base en una base ya encontrada:

Corolario 2.3.4 *Sea* $G = \{g_1, \ldots, g_t\}$ *una base de Gröbner para el ideal* I*. Para obtener una base de Gröbner mínima con base en* G*, eliminar todos los* g_i *para los cuales existe* $j \neq i$ *tal que* $\mathrm{lp}(g_j)$ *divide a* $\mathrm{lp}(g_i)$ *y finalmente, dividir cada* g_i *restante por el* $\mathrm{lc}(g_i)$*.*

Sin embargo, las bases de Gröbner mínimas no son únicas. Para encontrar una única base de Gröbner se requiere de:

Definición 2.3.14 *Una base de Gröbner* $G = \{g_1, \ldots, g_t\}$ *se conoce como base de Gröbner reducida si para todas las* i*,* $\mathrm{lc}(g_i) = 1$ *y* g_i *está reducida respecto a* $G - \{g_i\}$*. Es decir, para toda* i*, ningún término diferente de cero en* g_i *es divisible por algún* $\mathrm{lp}(g_j)$ *para cualquier* $j \neq i$*.*

Una base de Gröbner reducida es también mínima.

2.3.6 Complejidad

Como se puede revisar en (Buchberger, 2001), la complejidad del cálculo de una base de Gröbner, puede ser exponencial en relación al número de variables de los polinomios. Puede tomar mucho tiempo terminar y en el peor de los casos, no terminar por acabarse la memoria del equipo.

En (Bardlet, 2002) se estudian propiedades de las bases de Gröbner en relación a su complejidad y se realiza una revisión del algoritmo $F5$, que es uno de los mejores que hay para el cálculo de estas bases. Se menciona el problema de espacio exponencial requerido por el algoritmo original y también, como en la realidad, el algoritmo puede tener un desempeño mejor que ese comportamiento exponencial.

El tema de bases de Gröbner esta en constante estudio y nuevas mejoras a los algoritmos actuales surgen constantemente. Algunos de los más importantes son *F4* (Faugère, 1999), *XL* (Courtois et al., 2000), *F5* (Faugère, 2002), o bien una modificación a *F5* y *G2V* (Gao et al., 2010).

2.4 Pruebas de conocimiento nulo

Los pruebas de conocimiento nulo tienen como objetivo, probar una declaración sin producir nada adicional a que ésta es válida. Así, una prueba de conocimiento nulo es convincente y no produce nada más allá de la validez de la aseveración que se está probando. Por ejemplo, en el caso de un sistema de ecuaciones cuadráticas la declaración es que un *probador* tiene la solución a dicho sistema de ecuaciones y la

prueba de conocimiento nulo sería comprobar, ante un *verificador*, que se tiene esa solución sin entregar los valores de la solución. Para un tratamiento más profundo de este tema, refiérase a (Goldreich, 2008, Sec. 9.2), así como a (Goldreich, 2004, Cap. 4).

2.4.1 Definición

En un esquema de prueba de conocimiento nulo, podemos ver al *verificador* como un adversario que pretende obtener información adicional a la de sólo convencerse que el *probador* tiene realmente lo que dice tener. En relación al modelo de seguridad, un adversario no deberá obtener nada adicional a lo que un comportamiento *bueno* (aquel que se apega estrictamente al protocolo definido en la prueba de conocimiento nulo) puede obtener con la misma cantidad de recursos de cómputo.

En la siguiente definición, establecemos la seguridad perfecta, basada ésta en el hecho de que un *verificador* después de interactuar con un *probador*, usando una estratégia no necesariamente especificada por el protocolo, obtiene la misma salida que la de un algoritmo al que sólo se le da una entrada y genera una salida sin más búsqueda de información:

Definición 2.4.1 *Conocimiento nulo perfecto: la estrategia de un probador P, se dice que es de conocimiento nulo perfecto sobre un conjunto S, si por cada estratégia de un verificador en tiempo polinommial probabilístico*[9]*, V^*, existe un algoritmo en tiempo polinomial probabilístico*[10]*, A^*, tal que*

$$(P, V^*)(x) \equiv A^*(x), \text{ para cada } x \in S$$

Donde $(P, V^*)(x)$ es una variable aleatoria[11] *que representa la salida del verificador V^* después de interactuar con el probador P en una entrada común x y $A^*(x)$ es una variable aleatoria que representa la salida del algoritmo A^* en la entrada x.*

En el caso anterior $\equiv$ significa igualdad. Si permitimos que esta igualdad cambie a cierta cercanía estadística, entonces se da origen a la definición de conocimiento nulo cuasi-perfecto o conocimiento nulo estadístico. Este es el sentido que generalmente toma la frase de conocimiento nulo, y en ésta, similitud significa indistinguible computacionalmente. Con esto, tenemos la siguiente definición:

[9] Normalmente tiene sentido asociar *cálculos eficientes*, con *cálculos en tiempo polinomial probabilístico*, ya que estos últimos tienen una probabilidad de falla en dicho cálculos, despreciable.

[10] Un algoritmo en tiempo polinomial probabilístico (*probabilistic polynomail time algorithm*) o proceso aleatorio (*randomized process*) es aquel que, independientemente de las decisiones aleatorias que realiza (los *volados* que lance internamente), siempre se detendrá, después de un número de pasos polinomial en la longitud de la entrada. Entonces, el número de volados que el algoritmo en tiempo polinomial probabilistico A^* realiza, esta limitado por un polinomio, denotado como T_{A^*}, que depende de la longitud de la entrada. Sin perder generalidad, suponemos que en la entrada x, el algoritmo siempre realiza $T_{A^*}(|x|)$ volados.

[11] Una variable aleatoria tradicionalmente se define como una función del espacio de muestreo a los reales. Sin embargo, en temas de complejidad computacional y criptografía, se considera también el mapeo del espacio de muestreo al espacio de cadenas binarias. En este caso, el espacio de muestreo es el conjunto de todas las cadenas de longitud ℓ-bits y cada una de estas cadenas de bits tiene asignada una medida de probabilidad de $2^{-\ell}$.

Definición 2.4.2 *Conocimiento nulo: la estrategia de un probador P, se dice que es de conocimiento nulo sobre un conjunto S si por cada estrategia de un verificador en tiempo polinomial probabilístico, V^*, existe un simulador en tiempo polinomial probabilístico, A^*, tal que para cada diferenciador en tiempo polinomial probabilístico, D, se cumple que:*

$$d(n) \stackrel{def}{=} \max_{x \in S \cap \{0,1\}^n} \{|Pr[D(x,(P,V^*)(x)) = 1] - Pr[D(x,A^*(x)) = 1]|\}$$

es una función despreciable[12].

2.4.2 Conocimiento nulo en esquemas de polinomios

En el año 2011, se publicó el artículo (Sakumoto et al., 2011), donde se propone un esquema de identificación para clave pública, basado en el problema de resolver un sistema cuadrático de varias variables, empleando además el concepto de conocimiento nulo. La solución al sistema de ecuaciones cuadráticas x forma la clave privada (secreto) y el sistema de polinomios $y = F(x)$ será la clave pública. Considera la forma polar asociada $G(x,y)$ de $F(x)$: $G(x,y) = F(x+y) - F(x) - F(y)$ que es una función bilineal.

Para este esquema de prueba de conocimiento nulo, se establece primeramente lo siguiente:

1. $x = r_0 + r_1$
2. $y = F(r_0 + r_1) = G(r_0, r_1) + F(r_0) + F(r_1)$
3. $r_0 = t_0 + t_1$
4. $F(r_0) = e_0 + e_1$
5. Y como consecuencia: $y = G(t_0, r_1) + e_0 + F(r_1) + G(t_1, r_1) + e_1$[13]

Considerando estas características, la identificación con prueba de conocimiento nulo se realiza como se muestra en el Algoritmo 4.

En la línea 3, *Comm* se refiere a una función compromiso (*commitment*), misma que se puede realizar con base en una función picadillo resistente a colisiones. Esta función compromiso es estadísticamente de encubrimiento, es decir, para las duplas (u,v) y (u',v') elegidas uniformemente, entonces $Comm(u,v)$ y $Comm(u',v')$ son estadísticamente indistinguibles. Es decir, el compromiso de u prácticamente no revela información de x aún para un verificador muy poderoso. Por otra parte, esta misma

[12] Es decir, $d(n)$ desaparece más rápido que el reciproco de cualquier polinomio positivo p: $d(n) < \frac{1}{p(n)}$, donde n es lo suficientemente grande

[13] Aquí, se consideró la siguiente propiedad de una forma bilineal: $B(u+v,w) = B(u,w) + B(v,w) \forall u,v,w \in V$, donde V es un espacio vectorial.

función de compromiso es computacionalmente obligada, ya que la probabilidad de que $Comm(u, v) = Comm(u^{'}, v^{'})$ es despreciable.

Este protocolo corresponde a uno de conocimiento nulo estadístico, como se estableció en la Definición 2.4.2, debido a todos los datos que se intercambian entre el *probador* y el *verificador* y donde la información de la clave secreta es enviada, aunque dividida y enmascarada.

Algoritmo 4 Algoritmo para identificación con prueba de conocimiento nulo

Entrada: $x, F(x))$
Salida: Aceptación o rechazo de un *probador*
// **Actividades del *probador***:
Aleatoriamente seleccionar r_0, t_0, e_0
// Generar compromisos:
Calcular: $r_1 \leftarrow x - r_0, t_1 \leftarrow r_0 - t_0, e_1 \leftarrow F(r_0) - e_0$
Con una función compromiso calcular: $c_0 = Comm(r_1, G(t_0, r_1) + e_0)$, $c_1 = Comm(t_0, e_0), c_2 = Comm(t_1, e_1)$
Enviar c_0, c_1, c_2 al *verificador*
// **Actividades del *verificador***:
Seleccionar alguno de los 3 c_i que recibió e indica al *probador* cuál seleccionó
// **Actividades del *probador***: el *probador* responde al *verificador*, con los siguientes datos, según el compromiso c_i seleccionado:
if Seleccionó c_0 **then**
  Envía $\sigma = (r_0, t_1, e_1)$
else if Seleccionó c_1 **then**
  Envía $\sigma = (r_1, t_1, e_1)$
else
  Seleccionó c_2, envía $\sigma = (r_1, t_0, e_0)$
end if
// **Actividades del *verificador***: con base en la σ recibida y al c_i seleccionado:

if Había seleccionado c_0 **then**
  if $c_1 == Comm(r_0 - t_1, F(r_0) - e_1)$ **and** $c_2 == Comm(t_1, e_1)$ **then**
    OK
  end if
else if Había seleccionado c_1 **then**
  if $c_0 == Comm(r_1, y - F(r_1) - G(t_1, r_1) - e_1)$ **and** $c_2 == Comm(t_1, e_1)$ **then**
    OK
  end if
else
  if $c_0 == Comm(r_1, G(t_0, r_1) + e_0)$ **and** $c_1 == Comm(t_0, e_0)$ **then**
    OK
  end if
end if
if OK **then**
  return Acepta al *probador* como uno válido
else
  return Rechaza al *probador*.
end if

Capítulo 3

Funciones en un sentido y trampas

Una función en un sentido (*one way function*) es aquella que se puede encontrar fácilmente, pero que es difícil de invertir. Por otra parte, una trampa (*trapdoor*) nos permite invertir una función en un sentido. Veamos cada uno de estos conceptos más formalmente. En la Sección 3.1 revisaremos los conceptos básicos de la funciones en un sentido. En la Sección 3.2 veremos cómo con una trampa se pueden invertir funciones en un sentido y finalmente en la Sección 3.3 revisaremos todos los detalles de la construcción y seguridad del esquema Aceite-Vinagre No-Equilibrado.

3.1 Funciones en un sentido

Veamos la definición formal:

Definición 3.1.1 ***Función en un sentido:*** *Una función* $f : \{0,1\}^* \to \{0,1\}^*$ *se llama de un sentido, si las siguientes condiciones se cumplen:*

1. *Es fácil de calcular: existe un algoritmo en tiempo polinomial determinístico* A*, tal que en la entrada* x*, el algoritmo* A *entrega* $f(x)$*, es decir:* $A(x) = f(x)$*.*

2. *Difícil de invertir: para cada algoritmo en tiempo polinomial probabilístico* $A^{'}$*, cada polinomio positivo* $p(\cdot)$ *y todas las* $n's$ *suficientemente grandes,*

$$Pr[A^{'}(f(U_n), 1^n) \in f^{-1}(f(U_n))] < \frac{1}{p(n)}$$

U_n representa una variable aleatoria uniformemente distribuida en $\{0,1\}^n$ y entonces la probabilidad mencionada en el punto 2 se toma sobre todos los valores posibles de U_n y considerando todas las opciones aleatorias que tome $A^{'}$ con una distribución de probabilidad uniforme. Así mismo, esta definición no exige una pre-imagen específica de $f(x)$, cualguier valor que satisfaga $f^{-1}(f(x))$ será adecuado.

El término 1^n que se da al algoritmo $A^{'}$ corresponde a la longitud deseada de la salida. Esto se hace, para descartar la posibilidad de que una función se considere de

un sentido por tan sólo reducir su entrada drásticamente y entonces el algoritmo $A^{'}$ no tenga el tiempo para obtener la correspondiente imagen inversa [1].

3.2 Permutaciones trampa

Por otra parte, una permutación trampa en un sentido (*trapdoor one way permutations*), es una colección de permutaciones en un sentido $\{f_i\}$, con la propiedad adicional de que f_i se invierte fácilmente cuando se le da como una entrada auxiliar una trampa (*trapdoor*) para el índice i. La trampa para el índice i, que denotaremos como $t(i)$, no se puede calcular eficientemente a partir de i pero si se pueden calcular eficientemente las parejas $(i, t(i))$. Para definir formalmente una permutación trampa en un sentido, definiremos primero lo que es una colección de funciones en un sentido:

Definición 3.2.1 ***Colección de funciones en un sentido:***[2] *Sea D_i el dominio de la función f_i y $\bar{I}$ un conjunto infinito de índices. Entonces, una colección de funciones, $\{f_i : D_i \to \{0,1\}^*\}_{i\in\bar{I}}$ se conoce como de un sentido si existen tres algoritmos en tiempo polinomial probabilísticos I, D y F, tal que se cumplan las siguientes dos condiciones:*

1. *Fácil de muestrear y calcular: en la entrada de tamaño 1^n, la salida del algoritmo de selección de índice I, se distribuye en el conjunto $\bar{I} \cap \{0,1\}^n$ (i.e. es un índice de n-bits de longitud de alguna función). En la entrada $i \in \bar{I}$ (índice de una función), la salida del algoritmo D (algoritmo de muestreo del dominio) se distribuye sobre el conjunto D_i (i.e. sobre el dominio de la función f_i). En la entrada $i \in \bar{I}$ y $x \in D_i$, el algoritmo de evaluación F siempre entrega $f_i(x)$.*

2. *Difícil de invertir: para cada algoritmo en tiempo polinomial probabilístico, $A^{'}$, cada polinomio positivo $p(\cdot)$ y todas las n's suficientemente grandes,*

$$Pr[A^{'}(i, f_i(x)) \in f_i^{-1}(f_i(x))] < \frac{1}{p(n)}$$

   donde $i \leftarrow I(1^n)$ y $x \leftarrow D(i)$.

Se dice que la colección es un conjunto de permutaciones si cada una de las f_i's es una permutación sobre la correspondiente D_i y $D(i)$ está uniformemente distribuida en D_i.

Podemos ahora definir formalmente a una permutación trampa (*trapdoor permutation*) que es el nombre formal para estas funciones, aunque generalmente se les llama trampas nadas más (*trapdoors*):

[1] Recordar que un algoritmo de tiempo polinomial probabilístico siempre se detiene, esto después de un número de pasos polinomial en la longitud de la entrada. Revisar (Goldreich, 2004, pp. 32-33)

[2] Ver (Goldreich, 2008, Ap. C2)

Definición 3.2.2 *Una colección de permutaciones como se definió en 3.2.1, se conoce como permutación trampa si hay adicionalmente, dos algoritmos en tiempo polinomial probabilísticos $I^{'}$ y F^{-1} tal que:*

1. *La distribución $I^{'}(1^n)$ se extiende sobre pares de cadenas de manera que la primera cadena esté distribuida como en $I(1^n)$.*
2. *Para cada (i,t) en el rango de $I^{'}(1^n)$ y para cada $x \in D_i$ se cumple que $F^{-1}(t, f_i(x)) = x$.*

Es decir, t es una trampa que permite la inversión de f_i.

Con los conceptos formalizados de funciones en un sentido y permutación de trampas, revisemos a continuación, el esquema Aceite-Vinagre No-Equilibrado.

3.3 Esquema Aceite-Vinagre No-Equilibrado

El esquema Aceite-Vinagre No-Equilibrado (AVNE, con siglas originales *UOV: unbalanced oil vinegar*), fue desarrollado por Patarin (Kipnis et al., 1999). Corresponde a una modificación al método original conocido sólo como aceite-vinagre que se presentó en 1997, pero que en 1998 fue atacado como se ve en (Kipnis and Shamir, 1998). El método AVNE lleva su nombre por el hecho, de que los polinomios cuadráticos dividen el total de variables n en variables aceite o y variables vinagre v y como veremos más adelante, los términos cuadráticos se forman sólo por variables vinagre o bien por vinagre-aceite, pero nunca sólo por variables aceite (como en una ensalada, donde el aceite nunca se mezcla completamente con el vinagre). En el esquema equilibrado, el número de variables aceite es igual a las variables vinagre: $n = o + v, v = o$ y después del ataque arriba mencionado, se propuso que las variables vinagre deberían ser al menos el doble de las aceite: $v \geq 2o$. En (Kipnis et al., 1999) se encuentra una restricción más que se debe cumplir: si $v \geq \frac{o^2}{2}$ el esquema también es inseguro. Debido a lo anterior, en este trabajo usamos $v = \lceil 2o \rceil$. Han sido varios los análisis que presentan al método AVNE como vulnerable, ver por ejemplo (Wolf and Preneel, 2005b). Sin embargo, bajo los parámetros apropiados este método es aún seguro, como se menciona en (Barreto et al., 2013, p. 407).

3.3.1 Detalle del método Aceite-Vinagre No-Equilibrado

El esquema de la función en un sentido presentado en este documento, está basado en el uso de ecuaciones cuadráticas de varias variables. La Figura 3.1 muestra el esquema general de este método. Hagamos las siguientes consideraciones:

- Sean $n, v, o \in \mathbb{N}$ constantes, siendo n el número de variables de las ecuaciones cuadráticas en varias variables, v será el número de variables vinagre y o el número de variables aceite. El número total n de variables, se dividirá en las vinagre y las aceite: $n = v + o$.

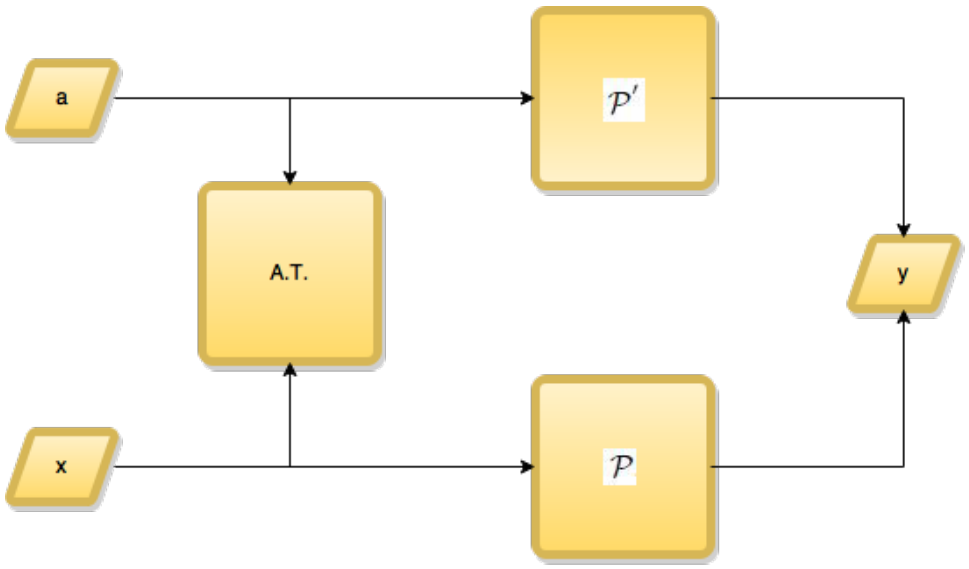

Figura 3.1: Esquema general del método Aceite-Vinagre No-Equilibrado

- $m \in \mathbb{N}$ representará el número de ecuaciones que se tendrán en el sistema cuadrático de ecuaciones. Por definición de este método, $m = o$ y del parrafo anterior: $v = n - m$.

- Sea $\mathcal{P}$ un conjunto de m polinomios cuadráticos en n variables. En este caso se denominarán a estas n variables como $\mathbf{x} = x_1, \ldots, x_n$

- Sea $\mathcal{P}'$ un conjunto de m polinomios cuadráticos en $n = v+o$ variables. Para este sistema de ecuaciones, se nombrarán a estas n variables como $\mathbf{a} = a_1, \ldots, a_v, a_{v+1}, \ldots, a_n$, donde las variables vinagre son $a_1, \ldots, a_v$ y las variables aceite son $a_{v+1}, \ldots, a_n$.

- Si se conocen las n variables $\mathbf{x}$, se pueden conocer los valores a los que evalúa cada una de los m polinomios que forman $\mathcal{P}$. Así mismo, $\mathcal{P}$ es invertible si conociendo los m valores de la imagen que toman los polinomios en $\mathcal{P}$, se pueden encontrar los valores que toman las n variables $\mathbf{x}$. Y de manera similar, invertir $\mathcal{P}'$ será encontrar los valores de las n variables $\mathbf{a}$ cuando se conoce la imagen de los m polinomios que forman $\mathcal{P}'$.

Como n es mayor a m y se trata de un sistema de polinomios cuadráticos, el sistema de ecuaciones $\mathcal{P}$ no es invertibles fácilmente, de hecho su solución está clasificada como NP-Completa (ver Sección 2.2.2). Por otra parte, el sistema de ecuaciones $\mathcal{P}'$ será fácilmente invertible ya que tiene una trampa formada por la división que se hace de las $n = v + o$ variables y la forma como se tratan éstas.

Para explicar porque $\mathcal{P}'$ es fácilmente invertible, explicaremos primero el bloque *A.T.* de la Figura 3.1 que es una transformación afín. La formamos por una matriz de orden $(n \times n)$ no singular M y un vector $\mathbf{w}$ de n elementos, que generamos de

manera aleatoria y que forman parte del *secreto*, es decir parte de la clave privada. Definimos a esta transformación afín como:

$$S(\mathbf{x}) = M \cdot \mathbf{x} + \mathbf{w} = \mathbf{a} \tag{3.1}$$

Así mismo y dado que la matriz M es invertible por haberse generado como no singular, podemos encontrar $\mathbf{x}$ en función de $\mathbf{a}$:

$$S^{-1}(\mathbf{x}) = M^{-1}(S(\mathbf{x}) - \mathbf{w}) = M^{-1}(\mathbf{a} - \mathbf{w}) = \mathbf{x} \tag{3.2}$$

Por otra parte, los polinomios que forman $\mathcal{P}^{'}$ los generaremos de la siguiente manera:

$$p_i(a_1, \ldots, a_n) = \sum_{j=1}^{n-m} \sum_{k=1}^{n} \gamma_{i,j,k} a_j a_k + \sum_{j=1}^{n} \beta_{i,j} a_j + \alpha_i \tag{3.3}$$

Por esta construcción, observamos, que las variables a_j de la primera doble sumatoria, son solamente las variables vinagre, ya que sus índices van desde $j = 1$ hasta $j = n - m = v$ y sin embargo se multiplican por las n variables (índice k), i.e. las variables vinagre son las únicas que podrán aparecer multiplicadas por otras vinagre, mientras que las variables aceite no aparecerán multiplicadas por otra variable aceite, sólo por otra vinagre. Concluimos entonces, que los polinomios en $\mathcal{P}^{'}$, son cuadráticos sólo en las variables vinagre.

Debido a esta última consideración, construimos al esquema AVNE de la siguiente manera:

- Generar aleatoriamente los m polinomios en n variables de $\mathcal{P}^{'}$, cumpliendo con las características antes mencionadas, particularmente con la Ecuación 3.3.
- Generar la matriz M y el vector $\mathbf{w}$ también aleatoriamente pero asegurando que la matriz M sea no singular y por lo tanto invertible.
- Construir ahora los m polinomios en n variables de $\mathcal{P}$. Para esto, obtener las variables a_i en función de las variables x_i, usando la transformación afín, ver Ecuación (3.1). Esto es, cada variable a_i es igual a la suma de los productos de los elementos de cada renglón de la matriz M por las variables x_i, más la componente w_i del vector $\mathbf{w}$: $a_i = \sum_{j=1}^{n} M_{i,j} x_j + w_i$. Posteriormente, estas variables a_i (que en este momento están representadas en función de las variables x_i), se sustituyen en los polinomios $\mathcal{P}^{'}$ y después de realizar su reducción simbólica, se obtienen las expresiones de la salida $\mathbf{y}$ (ver Figura 3.1). Como esta $\mathbf{y}$ debe ser igual a la salida que entreguen los polinomios $\mathcal{P}$, entonces estas m ecuaciones formarán $\mathcal{P}$.

Con este esquema, si se tienen los valores de $\mathbf{x} = x_1, \ldots, x_n$, se sustituyen dichos valores en $\mathcal{P}$ y se obtiene la salida $\mathbf{y}$ (observar que $\mathbf{y}$ tiene un tamaño de $m = o$ valores). Alternativamente, los valores $\mathbf{x}$ se pasan por la transformación afín y se genera el vector $\mathbf{a} = a_1, \ldots, a_n$, mismo que se sustituye en los polinomios $\mathcal{P}^{'}$ obteniéndose la

misma salida $\mathbf{y}$.

Por otra parte, y esto es lo que permite invertir $\mathcal{P}'$ fácilmente, si se proporciona la imagen $\mathbf{y}$ buscando encontrar la pre-imagen $\mathbf{x}$ que genera dicha salida bajo los polinomios $\mathcal{P}$, realizaremos lo siguiente:

- En el sistema de ecuaciones $\mathcal{P}'$, sustituir valores aleatorios en las variables vinagre, quedando entonces m ecuaciones *LINEALES* con $o = m$ incógnitas, ya que todas las variables vinagre se sustituyen por los valores numéricos generados aleatoriamente y por la forma en que se generó $\mathcal{P}'$ las variables aceite nunca aparecen multiplicadas por ellas mismas.

- Resolver el sistema de ecuaciones que se genera y con esto se conocerán las $n = v + o$ variables $\mathbf{a}$.

- Usando la transformación inversa afín, Ecuación 3.2, generar las variables $\mathbf{x}$ que entreguen el valor solicitado de la imagen $\mathbf{y}$.

- En caso que el sistema de ecuaciones generado, no tenga solución, se generan nuevos valores aleatorios para v y se repite el ciclo hasta encontrar las variables v, o.

3.3.2 La clave pública no muestra estructura de clave privada

Como expusimos en la Sección 2.2.1, para el caso de *ecuaciones cuadráticas en varias variables*, los polinomios $\mathcal{P}$ toman la siguiente forma:

$$p_i(x_1, \ldots, x_n) := \sum_{1 \leq j \leq k \leq n} \gamma_{i,j,k} x_j x_k + \sum_{j=1}^{n} \beta_{i,j} x_j + \alpha_i \tag{3.4}$$

donde m representa al número de ecuaciones, $1 \leq i \leq m$, $\gamma_{i,j,k}, \beta_{i,j}, \alpha_i \in \mathbb{F}$ y en nuestro trabajo se tomo $\mathbb{F}_2$. Generalmente a $\gamma_{i,j,k}$ se le conoce como coeficiente cuadrático, a $\beta_{i,j}$ como coeficiente lineal y a α_i como el coeficiente constante. En el caso de $\mathbb{F}_2$, $j < k$ ya que $x_i^2 = x_i$. Por otra parte, dichos polinomios se generan de los polinomios $\mathcal{P}'$ y la transformación afín (n representa el número de variables y $n-m$ es el número de variables vinagre), ver Ecuaciones 3.1, 3.2 y 3.3.

Dado que a $\mathcal{P}$ lo generamos usando la transformación afín S y el resultado de dicha transformación se aplica a $\mathcal{P}'$ entonces, se puede decir que $\mathcal{P} = \mathcal{P}' \circ S$. De esta forma, $\mathcal{P}$ parecerá aleatoria dependiendo principalmente de la transformación afín S, es decir, de la matriz M y del vector $\mathbf{w}$, ya que aunque $\mathcal{P}'$ se genera aleatoriamente, debe cumplir con la estructura de variables aceite-vinagre.

NOTA: si M fuera la matriz identidad y $\mathbf{w}$ el vector cero, entonces $\mathcal{P} = \mathcal{P}'$, lo cual sería una falta de seguridad fatal. Sin embargo, si se considera a M de orden

$(n \times n)$ y a $\mathbf{w}$ de tamaño n, esto puede ocurrir con una probabilidad individual de:

$$P_{M=I} = \frac{1}{2^{n^2}} \tag{3.5}$$

$$P_{\mathbf{w=0}} = \frac{1}{2^n} \tag{3.6}$$

Y la probabilidad de que la matriz M sea la identidad y el vector $\mathbf{w}$ sea el vector cero, simultáneamente, es:

$$P_{(M=I \ \& \ \mathbf{w=0})} = \frac{1}{2^{n^2}} \times \frac{1}{2^n} = \frac{1}{2^{n(n+1)}} \tag{3.7}$$

Este valor tiende a cero de una manera exponencial con el valor cuadrático de n, por lo que puede despreciarse (para una $n = 10$ esta probabilidad es de 7.7×10^{-34}!)

Si consideramos que M_i representa el i-ésimo renglón de la matriz M, $M_{i,j}$ representa la j-ésima columna del i-ésimo renglón de la matriz M, $u_i = \{j \mid M_{i,j} = 1, \ 1 \leq j \leq n\}$ y considerando además la Ecuación 3.1, las variables $\mathbf{a}$ se pueden expresar como ($\mathbf{w}$ se considera formado como $[w_1, \ldots, w_n]$)

$$\mathbf{a} = [a_1, \ldots, a_n] = [(\sum_{j \in u_1} x_j) + w_1, \ldots, (\sum_{j \in u_n} x_j) + w_n] \tag{3.8}$$

De esta Ecuación 3.8 vemos, que las variables $\mathbf{a}$, dependiendo de los valores que tenga M, podrán formarse a partir de $\mathbf{x}$ y por lo tanto $\mathcal{P}$ podrá tener entre sus términos cuadráticos a cualquier producto de variables $\mathbf{x}$ mezclándose éstas sin poder saber su relación con las variables aceite o vinagre. Esto se podría conocer si se descubrieran los valores de M y $\mathbf{w}$. Concluimos entonces, que aunque los polinomios en $\mathcal{P}'$ tienen una estructura definida por el hecho que las variables cuadráticas nunca se forman por dos variables aceite, los polinomios $\mathcal{P}$ no muestran esa estructura, ya que ellos por medio de la transformación afín, combinan a todas las variables $\mathbf{x}$ para formar los términos cuadráticos.

3.3.3 Ecuaciones simultáneas siempre con solución

Como podemos apreciar de la Ecuación 3.3, estos polinomios ayudan a invertir $\mathcal{P}'$, es decir, si conocemos los valores de p_i, podemos encontrar los valores de $a_1, \ldots, a_n$ que satisfagan $\mathcal{P}'$. Para esto, basta con sustituir valores aleatorios para las variables vinagre en dichas Ecuaciones 3.3 y el sistema resultante, será de o ecuaciones en las o variables aceite. Sin embargo, podría ocurrir, que al realizar la sustitución de dichos valores aleatorios, el sistema de ecuaciones no tuviera solución. En este caso, lo que haríamos es generar nuevos valores aleatorios para las variables vinagre, sustituirlas nuevamente en $\mathcal{P}'$ y se volvemos a intentar resolver ese nuevo sistema de ecuaciones.

Existen dos casos en los que el sistema de ecuaciones antes mencionado, no tiene solución única. El primero cuando se tenga un sistema compatible indeterminado (conjunto infinito de soluciones) y el segundo cuando se tenga un sistema incompatible

(sin solución). Con base en el Teorema de Rouché-Frobenius, un sistema de ecuaciones es compatible si y sólo si la matriz de coeficientes y la matriz ampliada tienen el mismo rango. Además, será determinado (solución única) si el rango encontrado es igual al número de incógnitas e indeterminado si el rango es menor a dicho número de incógnitas.

Por otra parte, el rango de una matriz es el número máximo de columnas (o renglones) que son linealmente independientes. Por el métodos de Gauss, podemos descartar un renglón si:

a. Todos sus coeficientes son cero

b. Hay dos lineas iguales

c. Una linea es proporcional a otra

d. Una linea es combinación lineal de otras

Entonces, para una matriz con coeficientes en $\mathbb{F}_2$ de orden $((m+1)\times m)$, considerando la matriz ampliada, tenemos las siguientes probabilidades:

Del inciso a., la probabilidad que al menos un renglón sea cero es $\frac{1}{2^{m+1}}$

Del inciso b., la probabilidad es $\frac{1}{2^{m+1}} \times \frac{1}{2^{m+1}} = \frac{1}{2^{2(m+1)}}$

Para el caso del inciso c., y considerando $\mathbb{F}_2$, se reduce al inciso b.

Y del inciso d., se presentaría al menos con la suma de dos lineas, es decir con una probabilidad de $\frac{1}{2^{m+1}} \times \frac{1}{2^{m+1}} = \frac{1}{2^{2(m+1)}}$

De los datos anteriores, la probabilidad P_{NCD} de que el sistema que se genere aleatoriamente no sea compatible determinado es:

$$\begin{aligned} P_{NCD} &= \frac{1}{2^{m+1}} + \frac{1}{2^{2(m+1)}} + \frac{1}{2^{2(m+1)}} \\ P_{NCD} &= \frac{2^{m+1}+2}{(2^{(m+1)})^2} = \frac{2(2^m+1)}{(2^{(m+1)})^2} \\ P_{NCD} &\approx \frac{2(2^m)}{(2^{(m+1)})^2} = \frac{2^{m+1}}{(2^{(m+1)})^2} \\ P_{NCD} &\approx \frac{1}{2^{m+1}} \end{aligned} \tag{3.9}$$

O bien, la probabilidad que sí se encuentre un sistema compatible determinado es aproximadamente:

$$1 - \frac{1}{2^{m+1}} \tag{3.10}$$

que mientras más grande sea m (número de ecuaciones) más cercano será a la unidad. De cualquier forma, si no se encontrara un sistema compatible determinado, simplemente habría que volverlo a intentar, para con una alta probabilidad (Ecuación 3.10) encontrar ahora un sistema compatible determinado.

3.3.4 Número de matrices no singulares en característica 2

Dado que la transformación afín mencionada en la Ecuación 3.1 es importante para ocultar los polinomios en $\mathcal{P}'$, y ésta requiere una matriz invertible, veamos ahora si es que el número de matrices invertibles es alto, para tenener entonces una probabilidad también alta de que al generar dicha matriz aleatoriamente ésta sea no-singular.

Sea $\mathbb{F}$ un campo finito con $q =\mid \mathbb{F} \mid$ elementos. Si consideramos matrices cuadradas de orden $(n \times n)$ entonces, dichas matrices tendrán inversa si y sólo si su determinante es distinto de cero. Por otra parte, una matriz cuadrada tiene su determinante igual a cero en estos casos:

a. Si dos filas o dos columnas son iguales

b. Si dos filas o dos columnas son proporcionales

c. Si una fila o columna es combinación lineal de dos o más de las restantes filas o columnas

d. Si todos los elementos de una fila son cero

Considerando lo anterior, tenemos entonces que para que una matriz cuadrada que se genere aleatoriamente sea no-singular, se debe cumplir:

1. El primer renglón de la matriz puede tomar q^n combinaciones diferentes, menos el vector cero $(q^n - 1)$

2. El segundo renglón puede tomar q^n combinaciones diferentes menos q veces las que no puede tomar el renglón uno (son q veces, porque son las combinaciones proporcionales que se pueden generar). De esta manera se eliminan las condiciones a., b. y c.. Serán entonces $q^n - q$ combinaciones diferentes

3. El tercer renglón puede tomar q^n combinaciones diferentes, menos q veces las que no puede tomar el renglón dos, es decir $q^n - q^2$

4. El n-ésimo renglón podrá tomar q^n combinaciones diferentes menos q veces las que no puede tomar el renglón anterior: $q^n - q^{n-1}$

El número de matrices no-singulares posibles será entonces el producto de todos los renglones, es decir:

$$NM_{NS} = (q^n - 1)(q^n - q)(q^n - q^2)\dots(q^n - q^{n-1}) = \prod_{i=0}^{n-1}(q^n - q^i) \tag{3.11}$$

Calculemos ahora, la probabilidad de obtener aleatoriamente una matriz no-singular de orden $n \times n$:

La Ecuación 3.11 la podemos expresar como:

$$NM_{NS} = [q^0(q^n - 1)][q^1(q^{n-1} - 1)][q^2(q^{n-2} - 1)]\cdots[q^{n-1}(q^1 - 1)]$$
$$NM_{NS} = q^{1+2+\cdots+(n-1)}(q^n - 1)(q^{n-1} - 1)\cdots(q^1 - 1)$$

Si particularizamos ahora para $\mathbb{F}_2$ (i.e. $q = 2$), que es el caso en el presente trabajo:

$$NM_{NS} = 2^{1+2+\cdots+(n-1)}(2^n - 1)(2^{n-1} - 1)\cdots(2^1 - 1)$$
$$NM_{NS} = 2^{\frac{(n-1)n}{2}}(2^n - 1)(2^{n-1} - 1)\cdots(2^1 - 1)$$

Ahora bien, $(2^n-1)(2^{n-1}-1)\cdots(2^1-1) \approx 0.2887(2^n)(2^{n-1})\cdots(2^1)$[3] $= 0.2887(2^{\frac{n(n+1)}{2}})$
Entonces:

$$NM_{NS} \approx 2^{\frac{(n-1)n}{2}}(0.2887)(2^{\frac{n(n+1)}{2}})$$
$$NM_{NS} \approx 0.2887(2^{n^2})$$

El número total de matrices diferentes que se puede generar en $\mathbb{F}_2$ es 2^{n^2} y por lo tanto, la probabilidad de generar aletoriamente una matriz no-singular es:

$$P_{M_{NS}} = 0.2887(2^{n^2})/2^{n^2} = 0.2887 \tag{3.12}$$

Concluimos que máximo después de 3.4 intentos, la matriz que se genere aleatoriamente será no-singular.

[3]Esto se obtuvo artiméticamente y es valido para $n \geq 15$

Capítulo 4

Protocolos propuestos

Considerando la teoría expuesta en la Sección 3.3, contamos con un esquema *AVNE*, que nos permite generar un grupo de polinomios que conformarán la clave pública $\mathcal{P}$ y a su vez, la clave privada formada por un grupo de polinomios adicionales $\mathcal{P}^{'}$ junto con una matriz de orden $(n \times n)$ no-singular M y un vector $\mathbf{w}$ de dimension n[1]. Pero aún más, este esquema *AVNE* nos permite invertir fácilmente $\mathcal{P}$[2], es decir $\mathbf{x} = \mathcal{P}^{-1}(\mathbf{x}) = \mathbf{y}^{-1}$ como se explicó en 3.3.

En la Sección 4.1 de este capítulo, analizaremos el protocolo de autenticación de conocimiento nulo basado en el esquema *AVNE* y considerando que dicho esquema es seguro, demostraremos que el protocolo propuesto también lo es. Esto mismo analizaremos en la Sección 4.2 para el protocolo de cifrado que se basa también en el esquema *AVNE* y que a su vez toma la idea empleada en el protocolo de autenticación que proponemos.

4.1 Protocolo de autenticación sobre el esquema AVNE

El esquema AVNE, es usado para generar firmas de mensajes. En esta sección, presentamos una opción, para usar el esquema AVNE, en un protocolo de autenticación con conocimiento nulo perfecto (ver Definición 2.4.1).

4.1.1 Descripción del protocolo de autenticación

Como vimos en la Sección 3.3, mediante el esquema AVNE, podemos tener una función en un sentido, que nos permita invertirla si conocemos la clave privada. Así, en el esquema de autenticación que proponemos, y como en todo esquema de reto/respuesta, interviene un *probador* y un *verificador*. En este caso, el *probador* generará primeramente el conjunto de polinomios $\mathcal{P}^{'}$, considerando su construcción como

[1]Recordar, que n representa el número de variables diferentes en cada polinomio

[2]En este caso $\mathcal{P}$ y $\mathcal{P}^{'}$ son m polinomios cuadráticos, tal como lo hemos venido manejando

se muestra en la Ecuación 3.3. Después genera la matriz no-singular M de $(n \times n)$ elementos en $\mathbb{F}_2$ así como el vector n-dimensional $\mathbf{w}$. Estos tres elementos forman la clave privada completa.

Con esta tripleta $(\mathcal{P}', M, \mathbf{w})$, se puede generar ahora la clave pública $\mathcal{P}$ como se estudió en la Sección 3.3.1. Estos polinomios $\mathcal{P}$ serán por definición, conocidos por cualquier entidad y desde luego por el *verificador* de nuestro protocolo.

El protocolo de autenticación de conocimiento nulo perfecto, que proponemos, es el siguiente:

1. Primeramente, el *verificador* genera aleatoriamente un vector $\mathbf{y} \in \mathbb{F}_2$ de dimension m con la finalidad de imponer un reto al *probador*: encontrar los valores de $\mathbf{x}$ tal que $\mathcal{P} : \mathbf{x} \mapsto \mathbf{y}$. Este vector $\mathbf{y}$ lo envía el *verificador* al *probador*.

2. Cuando el *probador* recibe el vector $\mathbf{y}$, sustituye valores aleatorios para las variables vinagre (variables v) en $\mathcal{P}'$ quedando un sistema de o ecuaciones en o incógnitas. Resuelve este sistema, para encontrar los valores de las variables aceite o y al unirlas con las variables vinagre v generadas aleatoriamente, se obtienen las n variables $\mathbf{a}$[3]. Aplica ahora la transformación inversa afín (Ecuación 3.2), encontrando así los valores de $\mathbf{x}$ que satisfacen a todos los polinomios $\mathcal{P}$. Cuando el *probador* termina estos cálculos, avisa al *verificador* que ya tiene los valores de $\mathbf{x}$[4], mediante una señal de *Listo*.

3. El verificador ahora escoge aleatoriamente un número $1 \leq i \leq m$. Este valor de i representa el número de polinomios que el *verificador* deberá tomar para realizar una combinación lineal con ellos. Escoge entonces i polinomios aleatoriamente del conjunto de polinomios $\mathcal{P}$, realizando la combinación lineal de estos. Llamaremos al polinomio resultante p_{LC}. Además, combina también los correspondientes valores del vector $\mathbf{y}$ (de acuerdo a los polinomios seleccionados antes) generando el valor al que debe evaluar p_{LC}; llamaremos a este valor y_{LC}.

4. Dado que el Algoritmo 2 podría usarse para encontrar que polinomios se usaron en la combinación lineal que produce p_{LC}, esto cuando se obtiene un residuo $r = 0$, el *verificador* se asegura que ésto no sucederá y para esto, revisa que el término principal de p_{LC} no aparezca en ninguno de los términos principales de los polinomios públicos $\mathcal{P}$. Si se tiene que el término principal de p_{LC} existe en alguno de los términos principales de $\mathcal{P}$, se genera otro p_{LC} al regresar al punto 3

5. El *verificador* envía al *probador* únicamente el nuevo polinomio generado p_{LC} pidiéndole el valor al que debe evaluar dicho polinomio, es decir el valor de y_{LC}.

[3]Recordar de lo expuesto en la Sección 3.3, que si el sistema de ecuaciones en las o variables que se obtiene al sustituir valores aleatorios para las variables v no tiene solución, lo que se debe hacer es sustituir nuevos valores aleatorios para v y volverlo a intentar; la probabilidad de encontrar soluciones para las variables o es muy alta, como se determinó en la Ecuación 3.10.

[4]Es importante recalcar para este punto, que el *probador* sólo **avisa** al verificador que ya tiene los resultados para $\mathbf{x}$, pero nunca le envía los mismos.

	probador		verificador
1		$\overset{y}{\leftarrow}$	$\mathbf{y} \overset{Ran}{\leftarrow} \{0,1\}^m$
2	Hallar $\mathbf{x}$ tal que $\mathcal{P} : \mathbf{x} \mapsto \mathbf{y}$	$\overset{Listo}{\rightarrow}$	
3			Escoge i polinomios de $\mathcal{P}$ y los combina linealmente (genera p_{LC}) junto con los valores correspondientes de $\mathbf{y}$ (y_{LC})
4			Revisa que el término principal de p_{LC} no sea igual a ninguno de los términos principales de $\mathcal{P}$. Si lo es, regresa a 3
5		$\overset{p_{LC}}{\leftarrow}$	Envía solamente p_{LC}
6	Calcula $y'_{LC} = p_{LC}(\mathbf{x})$	$\overset{y'_{LC}}{\rightarrow}$	
7			Si $y'_{LC} \neq y_{LC}$ registra esto Repetir $1 \rightarrow 6$ r veces y si al menos una $y'_{LC} \neq y_{LC}$ entonces *rechaza*

Tabla 4.1: Protocolo de autenticación de conocimiento nulo perfecto

6. El *probador* al recibir el polinomio p_{LC} y dado que éste es el producto de una combinación lineal del sistema de ecuaciones $\mathcal{P} = \mathbf{y}$ del cual él ya tiene la solución $\mathbf{x}$, entonces, sólo deberá sustituir los valores $\mathbf{x}$ en p_{LC} y enviar ese resultado que llamaremos y'_{LC} al *verificador*[5].

7. Cuando el *verificador* recibe la respuesta del *probador* compara y_{LC} con y'_{LC}. Si no son iguales, registra esto, y sigue con el protocolo. Si son iguales sigue con el protocolo, que con la finalidad de reducir la probabilidad de que un *probador* falso pueda engañar al *verificador*, consiste en repetir los puntos 1 a 6 un número r predefinido de veces, teniéndose entonces una probabilidad de adivinar todos los desafíos de $1/2^r$. Al final de estos ciclos, si hay registro de al menos un $y'_{LC} \neq y_{LC}$ el *verificador* *rechaza*, de otra forma *acepta*.

En la Tabla 4.1 se muestra un resumen del protocolo de autenticación de conocimiento nulo perfecto.

4.1.2 Autenticación de conocimiento nulo perfecto

Como se mencionó en la Definición 2.4.1, tendremos un esquema de conocimiento nulo perfecto, si obtenemos el mismo resultado cuando un *verificador-probador* interactuan con el obtenido por un simulador (en este caso el algoritmo A^*) sin necesidad de interactuar con un *probador*.

Si analizamos el protocolo resumido en la Tabla 4.1, vemos que el verificador, cuando genera p_{LC} y y_{LC}, ha determinado la respuesta que un probador le debe enviar

[5]Dado que estamos trabajando en $\mathbb{F}_2$ un probador falso tiene 1/2 de probabilidad de acertar al resultado correcto.

para demostrar que es auténtico; en este caso sabe de antemano que el probador debe enviar y_{LC}. Así, el algoritmo A^* puede efectivamente simular al *probador*, cuando realiza la combinación lineal antes mencionada y obtiene y_{LC}. Este algoritmo para pasar la prueba, sólo tiene que simular que regresa como respuesta del *probador* la misma y_{LC} que acaba de determinar.

El error en el que se puede incurrir en este caso es de cero y por eso, clasificamos al protocolo empleado en este trabajo, como de conocimiento nulo perfecto.

Es interesante comparar la afirmación anterior con la Definición 2.4.2, donde se puede permitir una diferencia (despreciable, pero existente) entre la probabilidad de que un *diferenciador* encuentre si está observando a un *verificador-probador* real o bien a un simulador que aparenta realizar todo el protocolo. En este caso, el diferenciador vería exactamente el mismo comportamiento, por lo que la diferencia expresada en la Ecuación 2.4.2 en este caso es cero, conduciéndonos entonces al esquema de conocimiento nulo perfecto.

4.1.3 Robustez del protocolo de autenticación

Dado que el esquema de autenticación de conocimiento nulo perfecto utiliza como base al método AVNE, dependemos primeramente en este método, para que este protocolo de autenticación sea robusto. El método AVNE, hasta donde tenemos conocimiento el día de hoy, no ha sido roto mientras el número de variables vinagre sea el doble de las de aceite: $v = 2o$. En (Kipnis and Shamir, 1998) se detalla el ataque que sufrió AVNE cuando $v = o$ y en (Kipnis et al., 1999) se analiza cuando $v \geq \frac{o^2}{2}$ dando como resultado también un esquema inseguro. Entonces, sólo hay que apegarse a usar $v = 2o$, lo que permanece como seguro hasta ahora.

Por parte del protocolo para la autenticación con conocimiento nulo perfecto, tenemos que la seguridad radica principalmente en qué polinomios del conjunto $\mathcal{P}$ son los empleados por el *verificador*, para combinarlos linealmente. Si esto se pudiera predecir, entonces un atacante podría realizar la combinación lineal de antemano y sin necesidad de conocer los valores de $\mathbf{x}$, como sí conocería los de $\mathbf{y}$, entonces realizaría exactamente el mismo procedimiento que el *verificador* y por lo tanto encontraría y_{LC}, que como intruso, mandaría al *verificador*, y con probabilidad de 1 acertaría. Esto se podría lograr, usando el Algoritmo 2 si los polinomios públicos $\mathcal{P}$ fueran una base de Gröbner, situación que no aplica en este caso, dado el origen aleatorio de estos polinomios.

Sin embargo y para asegurarse que siempre se genere un residuo diferente de cero (ver explicación al final de la Sección 2.3.3 del porque debe ser diferente de cero) se revisa que el término principal del polinomio generado no sea igual a ninguno de los términos principales de los polinomios que forman la clave pública. Así, desde el primer ciclo de la división, r tomará por lo menos el primer término principal del polinomio generado, eliminando así la posibilidad de encontrar qué polinomios se usaron para generar p_{LC}. Esto conlleva un costo adicional para generar el mencionado polinomio, dado que si cuando se eligen aleatoriamente los polinomios de la clave pública que

Vars.	Polinomios	Terms.				Prom. términos diferentes
15	5	3	3	4	3	3.25
20	6	2	4	3	3	3
32	10	7	3	3	4	4.25
48	16	4	4	5	4	4.25
64	21	4	4	4	5	4.25
96	32	3	4	6	6	4.75
128	42	6	6	5	8	6.25

Tabla 4.2: Cantidad de términos principales diferentes en los polinomios de la clave pública

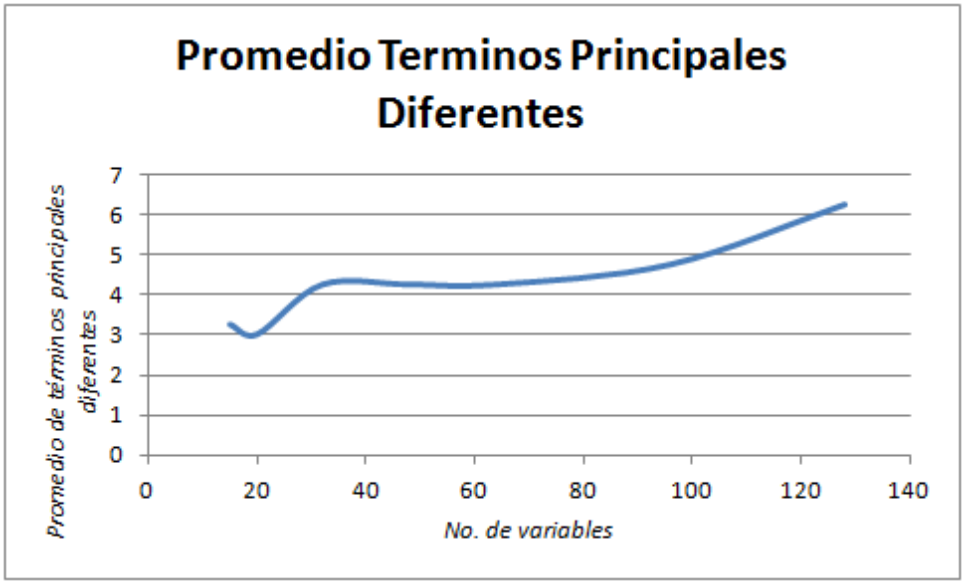

Figura 4.1: Promedio de términos principales diferentes en los polinomios de la clave pública

se combinarán para formar p_{LC}, se encuentra que su término principal es igual a alguno de los de la clave pública, se debe volver a intentar otra combinación. En la Tabla 4.2 se muestra el promedio del número de términos principales diferentes, en cuatro generaciones de polinomios, desde 15 hasta 128 variables. Así por ejemplo, el primer renglón es para cinco polinomios en quince variables, que tuvieron 3, 3, 4 y 3 términos principales diferentes entre los cinco polinomios y de estos cuatro datos tomamos el promedio, que se ve en la última columna. La Figura 4.1 muestra la gráfica del promedio de términos diferentes en los polinómios públicos vs. el número de variables de estos polinomios. Como se puede apreciar, aunque este número tiende a aumentar, este aumento es lento y su valor absoluto es pequeño comparado con el número de variables y por lo tanto, será muy probable que en pocas iteraciones se pueda encontrar el polinomio p_{LC} que cumpla con las características deseadas.

Adicionalmente a lo anterior, en cada ciclo del protocolo mostrado en la Tabla 4.1, se genera un nuevo vector $\mathbf{y}$ aleatoriamente y esto obligará a generar un nuevo poli-

nomio p_{LC} y nuevos cálculos habría que realizar para tratar de saber que polinomios generaron p_{LC}.

El número total de casos que tenemos para realizar las combinaciones lineales (NC) es:

$$NC = \binom{m}{1} + \binom{m}{2} + \cdots + \binom{m}{m} = \sum_{k=1}^{m} \binom{m}{k} \tag{4.1}$$

Esto porque de los m polinomios que forman la clave pública, podemos escoger uno de m, o bien dos de m y así sucesivamente hasta poder escoger m de los m polinomios.

La Ecuación 4.1, se puede reducir a:

$$NC = 2^m - 1 \approx 2^m \text{ para } m \text{ grandes.}$$

Proponemos como m grande a $m \geq 16$, ya que con eso la diferencia entre 2^m y $2^m - 1$ es $\leq 1.5 \times 10^{-3}$ %.

Por otra parte, para nuestro caso de estudio, $v = 2o$ y $m = o$, por lo que $n = o + v = 3o = 3m$.

Entonces, un parámetro de seguridad de 128 bits implica tener un sistema de 128 polinomios y para eso, $n = 3 \cdot 128 = 384$, i.e., debemos generar una clave pública de 128 polinomios en 384 variables. Observamos que este parámetro de seguridad hace que el número de variables de la clave pública se mueva linealmente.

4.2 Protocolo de cifrado sobre el esquema AVNE

Nuevamente, emplearemos el esquema AVNE, en este caso para generar un conjunto de polinomios $\mathcal{P}$ (clave pública) que nos permitirán cifrar un mensaje, y por medio de los polinomios $\mathcal{P}'$ y la transformación afín de este esquema, descifrar el mensaje antes cifrado. Esta operación se realizará bit a bit.

El protocolo de cifrado que describiremos en la Sección 4.2.1 cifra un bit a la vez y dado que lo hemos definido sobre el esquema asimétrico AVNE, podemos clasificarlo entonces como un cifrador de flujo (*stream cipher*) de clave pública.

El espacio de mensajes $\mathcal{M}$ será cualquier cadena de caracteres binarios: $\{0,1\}^*$. Por otra parte, usando el esquema AVNE se genera el conjunto de polinomios $\mathcal{P}$ que serán la clave pública pertenecientes al espacio formado por el conjunto de polinomios expresado en la Ecuación 2.5:

$$\mathcal{K}_{\mathcal{PK}} := \{ \sum_{1 \leq j \leq k \leq n} \gamma_{i,j,k} X_j X_k + \sum_{j=1}^{n} \beta_{i,j} X_j + \alpha_i \} \tag{4.2}$$

Necesitaremos también un espacio para las claves privadas (o claves secretas) que a diferencia de los métodos comunes (RSA, DH, etc.) es un espacio más grande que el de las claves públicas, en este caso será:

$$\mathcal{K}_{\mathcal{SK}} = \mathcal{K} \cup M_{n \times n} \cup \mathbf{w}_n \tag{4.3}$$

Donde $M_{n\times n}$ representa el conjunto de todas las matrices en $\mathbb{F}_2$ de orden $n \times n$ y $\mathbf{w}_n$ son todos los vectores también en $\mathbb{F}_2$ de dimension n.

El espacio de textos cifrados $\mathcal{C}$ será también el de polinomios cuadráticos en varias variables generado como se muestra en la Ecuación 4.2. La función de cifrado será $E : \mathcal{K}_{\mathcal{PK}} \times \mathcal{M}_i \rightarrow \mathcal{C}$ y la función de descifrado será $D : \mathcal{K}_{\mathcal{SK}} \times \mathcal{C} \rightarrow \mathcal{M}_i$ donde $\mathcal{M}_i$ representa sólo un bit del mensaje a cifrar.

Tenemos entonces todos los objetos para poder realizar un cifrado/descifrado asimétrico o bien, de clave pública:

- $\mathcal{M}$: un espacio de mensajes
- $\mathcal{K}_{PK}$ y $\mathcal{K}_{SK}$: un espacio de claves
- $\mathcal{C}$: un espacio de textos cifrados
- $E : \mathcal{K}_{\mathcal{PK}} \times \mathcal{M}_i \rightarrow \mathcal{C}$: una función de cifrado
- $D : \mathcal{K}_{\mathcal{SK}} \times \mathcal{C} \rightarrow \mathcal{M}_i$: una función de descifrado

4.2.1 Descripción del protocolo de cifrado

Emplearemos también aquí, el esquema AVNE como elemento base para construir sobre él un método de cifrado y descifrado de mensajes. En este caso, Alicia será quien genera la clave pública (conjunto de polinomios $\mathcal{P}$) y se la da a conocer a Beto que es quien desea enviar un mensaje cifrado. Así mismo, Alicia genera los elementos de la clave privada, es decir los polinomios $\mathcal{P}'$, la matriz no-singular de orden $(n \times n)$ M y el vector n-dimensional $\mathbf{w}$. Recordar que estamos trabajando en $\mathbb{F}_2$.

En este caso, el cifrado de mensajes se realiza bit a bit, contrastando con los métodos de uso común, que cifran por bloques. El protocolo de cifrado y descifrado es el siguiente:

1. Beto genera de manera aleatoria un vector $\mathbf{y} \in \mathbb{F}_2$ de dimension m, y lo envía a Alicia, para que ella encuentre los valores de $\mathbf{x}$ tal que $\mathcal{P} : \mathbf{x} \mapsto \mathbf{y}$.

2. Alicia recibe el vector $\mathbf{y}$ y busca los valores de $\mathbf{x}$ que satisfagan $\mathcal{P} : \mathbf{x} \mapsto \mathbf{y}$. Para esto, sustituye valores aleatorios en las variables vinagre v en el conjunto de polinomios secretos $\mathcal{P}'$ con lo que se obtiene un sistema de o ecuaciones en o variables, el cual debe poderse resolver de manera eficiente. En caso que no sea un sistema de ecuaciones compatible determinado (sistema con solución única) genera nuevamente valores aleatorios para las variables v, sustituye éstos valores en $\mathcal{P}'$ y determina si el nuevo sistema generado tiene solución, lo cual con una probabilidad alta sucederá (ver Ecuación 3.10). Las n variables $\mathbf{a}$ son las v variables generadas aleatoriamente más las o variables obtenidas al resolver el sistema de ecuaciones antes mencionado. Para encontrar los valores de las variables $\mathbf{x}$, se aplica la transformación inversa afín de la Ecuación 3.2 y estos valores son los que harán $\mathcal{P} = \mathbf{y}$. Al terminar esas operaciones, Alicia avisa a Beto que esta lista para continuar con el protocolo.

3. Beto ahora enviará un bit del mensaje $T = T_0, \ldots, T_k$, en este caso $T_j, 0 \leq j \leq k$. Para esto, escoge aleatoriamente un número $1 \leq i \leq m$. El valor de i representa el número de polinomios que Beto escogerá aleatoriamente del conjunto $\mathcal{P}$, los combina ahora linealmente y genera un nuevo polinomio p_{LC}. Realiza también la combinación lineal de los valores correspondientes de $\mathbf{y}$ generando y_{LC}. Revisa ahora que $y_{LC} == T_j$ y si no es el caso, repite el proceso de este punto hasta que se cumpla la igualdad anterior. Cuando esto sucede, tenemos un polinomio p_{LC} que representa el cifrado del bit T_j que se desea enviar.

4. Dado que el Algoritmo 2 podría usarse para encontrar que polinomios se usaron en la combinación lineal que genera p_{LC} esto cuando se obtiene un residuo $r = 0$, *Beto* se asegura que ésto no sucederá, revisando que el término principal de p_{LC} no aparezca en ninguno de los términos principales de los polinomios públicos $\mathcal{P}$. Si se tiene que el término principal de p_{LC} existe en alguno de los términos principales de $\mathcal{P}$, se genera otro p_{LC} por medio de regresar al punto 3

5. Beto envía únicamente el polinomio generado p_{LC} a Alicia.

6. Alicia recibe el polinomio p_{LC} y sustituye en él los valores que previamente había calculado de $\mathbf{x}$, obteniendo así el valor original del bit j del mensaje enviado. Con esto, Alicia ha descifrado el bit T_j a partir del polinomio recibido p_{LC}.

7. Si hay más bits por cifrar, se regresa al punto 1, hasta mandar cifrados, todos los bits de T.

Este protocolo de forma resumida, se presenta en la Tabla 4.3.

4.2.2 Robustez del protocolo de cifrado

Como en el caso del protocolo de autenticación (ver Sección 4.1.3), la seguridad básica de este protocolo de cifrado radica en el esquema AVNE. Consideremos por las explicaciones propias de este método vistas en la Sección 3.3 que cuando $v = 2o$ el método es seguro. Analicemos entonces cuál es la seguridad que ofrece este método de cifrado por el protocolo mencionado en la Sección 4.2.1.

Primero, consideremos que en una palabra de m bits, tenemos 2^m combinaciones diferentes, cada una representando a un número binario desde 0 hasta $2^m - 1$. De estos 2^m números, exactamente la mitad, serán números pares y la otra mitad impares, y la combinación lineal[6] de los bits módulo 2 que representan cada número binario resultará en 2^{m-1} ceros e igual cantidad de unos (ver la Tabla 4.4 como un ejemplo para $m = 4$).

Por otra parte, si de estos m bits tomamos uno a la vez, tendremos m combinaciones diferentes, y en este caso estas combinaciones corresponden con el número binario de donde se están tomando dichas combinaciones (ver por ejemplo la Tabla 4.5). Si observamos esta tabla, tenemos que la probabilidad de encontrar un uno (resp. cero) es 1/2 (resp. 1/2) y esto en cualquiera de las combinaciones que se seleccionen.

[6]Es decir la operación XOR de todos los bits en cada renglón

	Alicia		Beto
1		$\overset{y}{\leftarrow}$	$\mathbf{y} \overset{Ran}{\leftarrow} \{0,1\}^m$
2	Hallar $\mathbf{x}$ tal que $\mathcal{P} : \mathbf{x} \mapsto \mathbf{y}$	$\overset{Lista}{\rightarrow}$	
3			Escoge aleatoriamente i polinomios de $\mathcal{P}$ y los combina linealmente (genera p_{LC}) junto con los valores correspondientes de $\mathbf{y}$ (genera y_{LC}). Debe cumplirse que $y_{LC} == T_j$ i.e. y_{LC} debe ser igual al valor del bit a enviar. Si con los i polinomios seleccionados, esto no se cumple, repite este punto hasta que $y_{LC} == T_j$
4			Revisa que el término principal de p_{LC} no sea igual a ninguno de los términos principales de $\mathcal{P}$. Si lo es regresa a 3
5		$\overset{p_{LC}}{\leftarrow}$	Envía solamente p_{LC}
6	Calcula $T_j = p_{LC}(\mathbf{x})$		
7			Si hay más bits de T por cifrar regresa a 1

Tabla 4.3: Protocolo de cifrado de mensajes bit a bit. En este caso, Beto desea enviar un mensaje T dividido en bits $T_0, \ldots, T_k$

a	b	c	d	XOR
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Tabla 4.4: Combinación lineal de cada renglón para el caso de palabras de 4 bits

a	b	c	d	C1	C2	C3	C4
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	1
0	1	1	0	0	1	1	0
0	1	1	1	0	1	1	1
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	1
1	0	1	0	1	0	1	0
1	0	1	1	1	0	1	1
1	1	0	0	1	1	0	0
1	1	0	1	1	1	0	1
1	1	1	0	1	1	1	0
1	1	1	1	1	1	1	1

Tabla 4.5: Combinaciónes posibles para el caso de palabras de 4 bits tomando un bit a la vez: $\binom{4}{1} = 4$

Como un ejemplo adicional, si de estos cuatro bits tomamos dos (resp. tres) a la vez, tendremos $\binom{4}{2} = 6$ (resp. $\binom{4}{3} = 4$) combinaciones diferentes, ver Tabla 4.6 (resp. Tabla 4.7). Y nuevamente, si observamos esta tabla, tenemos que la probabilidad de encontrar un uno (resp. cero) es 1/2 (resp. 1/2) y esto en cualquiera de las combinaciones que se seleccionen (y de cualquiera de las dos tablas).

Regresando al caso de la robustez del cifrado presentado en la Sección 4.2.1, m representa el número de ecuaciones de la clave pública y como hemos visto de la discusión anterior, cuando se haya seleccionado aleatoriamente un $i, 1 \leq i \leq m$ y se realicen las combinaciones lineales de i polinomios también seleccionados al azar en $\mathcal{P}$ (generando entonces p_{LC}) así como con las respectivas $\mathbf{y}$, la probabilidad de que $y_{LC} = T_j$[7] es de 1/2.

La seguridad del protocolo radica principalmente en cuáles polinomios del conjunto $\mathcal{P}$ son los empleados por *Beto*, para combinarlos linealmente. Para este punto, referimos al lector a la Sección 4.1.3, donde se da una explicación del mecanismo desarrollado para evitar que se encuentren los polinomios de la clave pública que se usaron para generar p_{LC}.

Adicionalmente a lo anterior, en cada ciclo del protocolo mostrado en la Tabla 4.3, se genera un nuevo vector $\mathbf{y}$ aleatoriamente y esto obligará a generar un nuevo polinomio p_{LC} y nuevos cálculos habría que realizar para tratar de saber que polinomios generaron p_{LC}.

[7]Es decir, que al combinar linealmente las y_i seleccionadas y que producen y_{LC} éste bit sea igual al bit T_j que se desea enviar cifrado en un polinomio p_{LC}.

a	b	c	d	C1	C2	C3	C4	C5	C6
0	0	0	0	00	00	00	00	00	00
0	0	0	1	00	00	01	00	01	01
0	0	1	0	00	01	00	01	00	10
0	0	1	1	00	01	01	01	01	11
0	1	0	0	01	00	00	10	10	00
0	1	0	1	01	00	01	10	11	01
0	1	1	0	01	01	00	11	10	10
0	1	1	1	01	01	01	11	11	11
1	0	0	0	10	10	10	00	00	00
1	0	0	1	10	10	11	00	01	01
1	0	1	0	10	11	10	01	00	10
1	0	1	1	10	11	11	01	01	11
1	1	0	0	11	10	10	10	10	00
1	1	0	1	11	10	11	10	11	01
1	1	1	0	11	11	10	11	10	10
1	1	1	1	11	11	11	11	11	11

Tabla 4.6: Combinaciónes posibles para el caso de palabras de 4 bits tomando dos bits a la vez: $\binom{4}{2} = 6$

a	b	c	d	C1	C2	C3	C4
0	0	0	0	000	000	000	000
0	0	0	1	000	001	001	001
0	0	1	0	001	000	010	010
0	0	1	1	001	001	011	011
0	1	0	0	010	010	000	100
0	1	0	1	010	011	001	101
0	1	1	0	011	010	010	110
0	1	1	1	011	011	011	111
1	0	0	0	100	100	100	000
1	0	0	1	100	101	101	001
1	0	1	0	101	100	110	010
1	0	1	1	101	101	111	011
1	1	0	0	110	110	100	100
1	1	0	1	110	111	101	101
1	1	1	0	111	110	110	110
1	1	1	1	111	111	111	111

Tabla 4.7: Combinaciónes posibles para el caso de palabras de 4 bits tomando tres bits a la vez: $\binom{4}{3} = 4$

El número total de casos que tenemos para realizar las combinaciones lineales (NC) es:

$$NC = \frac{\binom{m}{1} + \binom{m}{2} + \cdots + \binom{m}{m}}{2} = \frac{1}{2}\sum_{k=1}^{m}\binom{m}{k} \tag{4.4}$$

Esto porque de los m polinomios que forman la clave pública, podemos escoger uno de m, o bien 2 de m y así sucesivamente hasta poder escoger m de los m polinomios, pero dado que sólo la mitad de ellos (probabilidad de que al combinarlos linealmente el resultado de y_{LC} sea igual a T_j) son realmente útiles, dividimos por dos.

La Ecuación 4.4, se puede reducir a:

$$NC = \frac{2^m - 1}{2} \approx 2^{m-1} \text{ para } m \text{ grandes.}$$

Proponemos como m grande a $m \geq 16$, ya que con eso la diferencia entre 2^{m-1} y $2^{m-1} - 0.5 \leq 1.5 \times 10^{-3}\,\%$.

Por otra parte, para nuestro caso de estudio, $v = 2o$ y $m = o$, por lo que $n = o + v = 3o = 3m$.

Entonces, un parámetro de seguridad de 128 bits implica tener un sistema de $128 + 1 = 129$ polinomios y para eso, $n = 3 \cdot 129 = 387$, i.e. debemos generar una clave pública de 129 polinomios en 387 variables. Observamos que este parámetro de seguridad hace que el número de variables de la clave pública se mueva linealmente.

Capítulo 5

Implementación y resultados obtenidos

Como mencionamos en el Capítulo 4, tanto el protocolo de autenticación como el de cifrado que proponemos en este trabajo, toman como esquema base al método AVNE (Aceite Vinagre No-Equilibrado). Empezaremos entonces, en la Sección 5.1, por describir como se implementó dicho esquema y la forma en que se representan los polinomios en varias variables que se usarán posteriormente. En la Sección 5.2, continuaremos con los detalles de la generación de claves y su administración. Finalmente, en la Sección 5.3, revisaremos los programas que implementan los protocolos de autenticación y cifrado.

Debemos considerar que en el esquema de AVNE, se tiene que realizar tanto una artimética de campos, en particular $\mathbb{F}_2$, así como el manejo de matrices y vectores. Debido a esto, y con la finalidad de concentrarnos principalmente en el concepto de los protocolos objetos de este trabajo, más que en la eficiencia de los programas desarrollados, éstos los implementamos en Sage (sin embargo, en el Apéndice A se presentan los programas que realizamos en `C++` para la generación de llaves privadas y públicas, mismas que se pueden generar en cualquier campo F_q, donde q es un número primo). Por otra parte, los programas para la codificación de las claves (ASN.1 y Base64) así como las decodificaciones respectivas, los implementamos en C. Con el objeto de encontrar la solución al sistema de ecuaciones que se generan con los polinomios que forman la clave pública (junto con el vector al que deben evaluar esos polinomios), realizamos pruebas para polinomios de diferentes número de variables, usando un programa especialmente para el caso de $\mathbb{F}_2$; dicho programa es PolyBoRi con el cual encontramos las bases de Gröbner para polinomios de hasta 19 variables.

Particularmente, la versión de Sage con la que trabajamos es: *SageMath Version 6.6, Release Date: 2015-04-14*, junto con Python versión *Python 2.7.6*. Para los programas en C, utilizamos el compilador *gcc* versión *4.8.2* y para los desarrollados en *C++* empleamos el compilador *g++* versión *4.8.2*. El desarrollo completo se realizó en Linux, particularmente la distribución *Ubuntu 14.04 ver. 3.13.0-53-generic*. Todos los programas que desarrollamos, tanto en Sage como en C y C++, aceptan en la misma

Directorio	Contenido
`3SAT`	Programa para la generación de instancias 3SAT
`ASN.1`	Programas para codificar y decodificar un archivo con polinomios representados en forma entera
`AuthProt`	Implementación del protocolo de autenticación bajo el esquema de cliente-servidor
`Base64`	Programas para codificar y decodificar un archivo en formato ASN.1
`data`	Directorio para almacenar claves en general
`DemoGenKeysCpp`	Programas en `C++` para generar las claves privadas y públicas en cualquier campo de orden primo
`EncProt`	Implementación del protocolo de cifrado-descifrado bajo el esquema de cliente-servidor
`files`	Directorio para almacenar claves en sus diferentes representaciones, producto de los programas en shell
`GroebnerBasis`	Programa para encontrar soluciones al sistema de polinomios públicos generados para el esquema AVNE
`Keys`	Programas para generar claves
`VeriProv`	Programas ejemplo de generación de estructuras de datos para el esquema de autenticación

Tabla 5.1: Directorios de programas y datos en `Programas`

linea de comandos, los parámetros con los que van a correr y regresan un código de error, en caso que esto ocurra. Esto nos permitió realizar un par de programas en el interpretador de comandos nativo de Linux: Bash. Estos programas tienen como finalidad realizar el ciclo completo de generación de claves. En relación al programa PolyBoRi empleado en las pruebas con bases de Gröbner, usamos la versión *PolyBoRi 0.8.3*.

En `Programas`[1], se encuentran los programas fuente y compilados, de forma que se puedan correr inmediatamente en un equipo de cómputo que cuente con las versiones de Sage, Python, gcc, g++, Linux y PolyBoRi mencionadas antes. Los directorios con los programas y datos en `Programas` son los mostrados en la tabla 5.1[2]

En este capítulo, explicaremos los programas que hay en los directorios antes mencionados, los requisitos para correr estos programas, las restricciones que tenemos

[1]Es importante observar, que esta palabra es propiamente un enlace y al dar clic sobre ella, nos llevará a una página donde aparecerá la estructura de directorios y archivos mostrada en la Tabla 5.1, mismos que se podrán accesar directamente al dar un clic sobre los nombres de estos directorios o archivos. Solamente los archivos que representan un programa ejecutable o bien los archivos compactados que contienen algunos datos (compactados para ahorrar espacio) se tendrán que descargar para ejecutarse o bien consultarse.

[2]Observar que como el desarrollo efectuado en este trabajo, es bajo un ambiente tipo Unix, las letras mayúsculas y minúsculas sí importan.

Programa	Función
`cryptokeys.sage`	Biblioteca de funciones para cifrar y descifrar archivos
`genkey.sage`	Programa para generar un archivo cifrado que contiene las claves privada y pública
`genkey.sage.py`	Archivo generado automáticamente por Sage, adecuando genkey.sage
`getkeys.sage`	Programa para obtener en archivos ASCII, las claves privada y pública
`getkeys.sage.py`	Archivo generado automáticamente por Sage, adecuando getkey.sage
`hashCL.py`	Programa que por linea de comando, genera un resumen (digest) de un archivo con la clave pública
`hash.py`	Programa interactivo que genera un resumen (digest) de un archivo con la clave pública
`uov.sage`	Biblioteca de funciones, para el esquema AVNE

Tabla 5.2: Contenido del directorio `Keys` del enlace `Programas`

en ellos y el orden en que estos programas se deben correr. Explicaremos también un ejemplo de la implementación de los protocolos de autenticación y de cifrado, mismos que dentro de un mismo equipo, los implementamos en un esquema cliente-servidor.

5.1 Implementación AVNE

Como mencionamos antes, el esquema AVNE explicado ampliamente en 3.3, lo desarrollamos en Sage. En el directorio `Keys` del enlace `Programas`, se encuentran los programas mostrados en la Tabla 5.2.

Los programas `cryptokeys.sage` y `uov.sage`, son bibliotecas, con varias funciones que son cargadas por los programas `genkey.sage` y `getkey.sage`. En `cryptokeys.sage` se tienen funciones que sirven para cifrar y descifrar un archivo, que en nuestro caso, corresponderá a un archivo con la clave privada. El algoritmo de cifrado es AES128 y para generar la clave usada en el cifrado y descifrado, se utiliza la función picadillo SHA256. Por otra parte, la biblioteca `uov.sage` contiene todas las funciones que permiten manejar la aritmética de campos (en este caso $\mathbb{F}_2$), toda la lógica para generar las claves privada y pública, la implementación de los algoritmos para representar con números enteros los polinomios $\mathcal{P}$, $\mathcal{P}'$ así como la matriz M_s y el vector $\mathbf{w}$. La biblioteca `uov.sage` contiene además funciones necesarias para la implementación de los protocolos de autenticación y cifrado, así como para escribir a archivos planos los polinomios generados y para leer esos mismos archivos y convertirlos a polinomios, con los que se pueda trabajar propiamente como polinomios y no como una cadena de caracteres leída de un archivo.

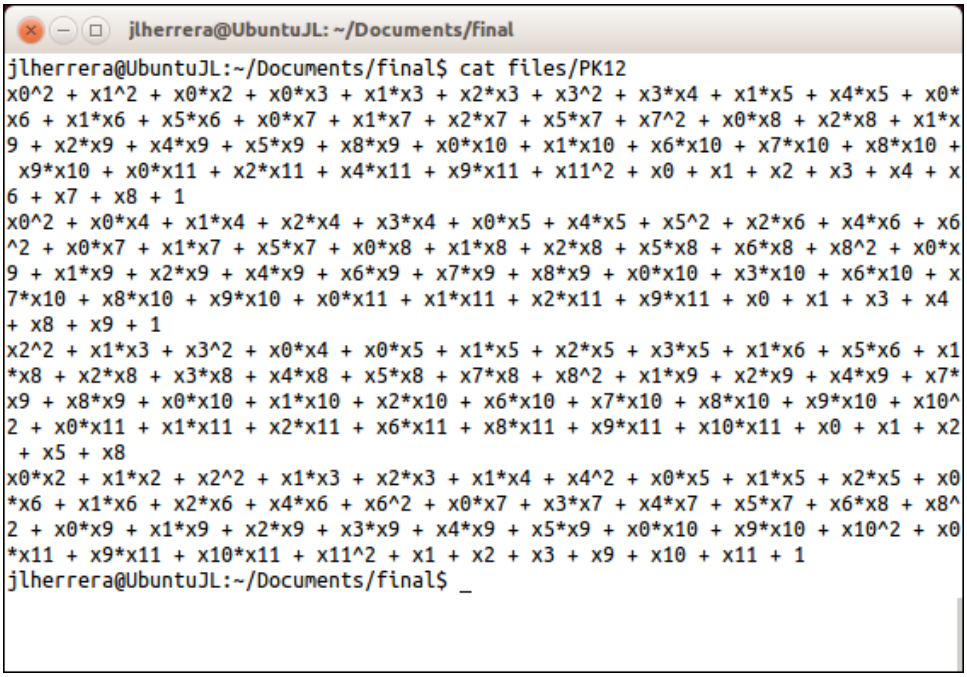

Figura 5.1: Cuatro polinomios cuadráticos en doce variables

5.1.1 Representación tipo enteros

En la Figura 5.1, mostramos el contenido de un archivo generado para doce variables ($n = 12$) y por lo tanto cuatro ($o = 4$) polinomios ($n = v + o = 2o + o = 3o$, i.e. $o = n/3$, ver Sección 3.3).

Como se puede apreciar, después del comando `cat files/PK12` que provoca se muestre el contenido del archivo PK12, las primeras cinco lineas, representan el primer polinomio, y así, cada cinco lineas representan otro polinomio, dando un total de cuatro polinomios. A este formato que muestra de forma explícita a los polinomios, le llamaremos la *representación natural de un polinomio*. Las variables van de $x0$ a $x11$ y cuando una variable aparece al cuadrado, por ejemplo para la variable $x0$, se representa como `x0^2`. Por otra parte, en cada polinomio, aparecen primero los términos cuadráticos, ya sea por una sola variable al cuadrado o por la multiplicación de dos variables; luego aparecen los términos lineales, es decir una sola variable y al final puede aparecer la constante 1 o bien no aparecer este término.

El problema con un archivo de este tipo, es que puede ocupar mucho espacio para el caso de más variables ya que habrá más monomios en cada uno de los polinomios, ver Ecuación 2.6a. Buscamos entonces una forma más compacta para representar cada polinomio y utilizamos las formas cuadráticas para representar un polinomio y dado que estamos trabajando en $\mathbb{F}_2$, esa representación en forma cuadrática la tomamos como un número binario que posteriormente se convierte a entero y a esta forma de ver cada polinomio, le llamamos representación entera. Veamos un ejemplo:

Supongamos que queremos representar el siguiente polinomio en 6 variables a su

representación entera:

$$p = x_0x_1 + x_1x_2 + x_2^2 + x_0x_3 + x_1x_3 + x_1x_4 + x_2x_4 + x_3x_4 + x_4^2 + x_0x_5 + x_3x_5 + x_5^2 + x_1 + x_3 + x_5 + 1 \quad (5.1)$$

Realizaremos la conversión en tres partes, primero los términos cuadráticos, luego los términos lineales y al final la constante.

1. Términos cuadráticos. Los términos involucrados son los siguientes:

$$p_{SQ} = x_0x_1 + x_1x_2 + x_2^2 + x_0x_3 + x_1x_3 + x_1x_4 + x_2x_4 + x_3x_4 + x_4^2 + x_0x_5 + x_3x_5 + x_5^2 \quad (5.2)$$

El polinomio de la Ecuación 5.2, lo podemos representar en su forma cuadrática, como se muestra en la Ecuación 5.3. La matriz M_{SQ} es una matriz con elementos en $\{0, 1\}$ de orden $n \times n$[3].

$$p_{SQ} = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 & x_4 & x_5 \end{bmatrix} \cdot M_{SQ} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \quad (5.3)$$

Si los elementos de la matriz M_{SQ} son $\{m_{i,j} | 0 \leq i, j \leq n-1\}$ entonces, la Ecuación 5.3 la podemos expresar como se ve en 5.4.

$$p_{SQ} = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 & x_4 & x_5 \end{bmatrix} \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} & m_{0,4} & m_{0,5} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} & m_{1,5} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} & m_{2,5} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} & m_{3,5} \\ m_{4,0} & m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} & m_{4,5} \\ m_{5,0} & m_{5,1} & m_{5,2} & m_{5,3} & m_{5,4} & m_{5,5} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \quad (5.4)$$

El producto de la matriz M_{SQ} con el vector $[x_0\ x_1\ x_2\ x_3\ x_4\ x_5]^T$, generará un vector como el que se muestra en la Ecuación 5.5.

$$\begin{bmatrix} m_{0,0}x_0 + m_{0,1}x_1 + m_{0,2}x_2 + m_{0,3}x_3 + m_{0,4}x_4 + m_{0,5}x_5 \\ m_{1,0}x_0 + m_{1,1}x_1 + m_{1,2}x_2 + m_{1,3}x_3 + m_{1,4}x_4 + m_{1,5}x_5 \\ m_{2,0}x_0 + m_{2,1}x_1 + m_{2,2}x_2 + m_{2,3}x_3 + m_{2,4}x_4 + m_{2,5}x_5 \\ m_{3,0}x_0 + m_{3,1}x_1 + m_{3,2}x_2 + m_{3,3}x_3 + m_{3,4}x_4 + m_{3,5}x_5 \\ m_{4,0}x_0 + m_{4,1}x_1 + m_{4,2}x_2 + m_{4,3}x_3 + m_{4,4}x_4 + m_{4,5}x_5 \\ m_{5,0}x_0 + m_{5,1}x_1 + m_{5,2}x_2 + m_{5,3}x_3 + m_{5,4}x_4 + m_{5,5}x_5 \end{bmatrix} \quad (5.5)$$

[3] n = número de variables, para este ejemplo $n = 6$

Este vector, al multiplicarse finalmente por $[x_0\ x_1\ x_2\ x_3\ x_4\ x_5]$, generará los términos cuadráticos. Esto lo podemos ver en la Ecuación 5.6.

$$\begin{aligned} p_{SQ} = & \\ & \begin{aligned} & m_{0,0}x_0x_0 + m_{0,1}x_0x_1 + m_{0,2}x_0x_2 + m_{0,3}x_0x_3 + m_{0,4}x_0x_4 + m_{0,5}x_0x_5 + \\ & m_{1,0}x_0x_1 + m_{1,1}x_1x_1 + m_{1,2}x_1x_2 + m_{1,3}x_1x_3 + m_{1,4}x_1x_4 + m_{1,5}x_1x_5 + \\ & m_{2,0}x_0x_2 + m_{2,1}x_1x_2 + m_{2,2}x_2x_2 + m_{2,3}x_2x_3 + m_{2,4}x_2x_4 + m_{2,5}x_2x_5 + \\ & m_{3,0}x_0x_3 + m_{3,1}x_1x_3 + m_{3,2}x_2x_3 + m_{3,3}x_3x_3 + m_{3,4}x_3x_4 + m_{3,5}x_3x_5 + \\ & m_{4,0}x_0x_4 + m_{4,1}x_1x_4 + m_{4,2}x_2x_4 + m_{4,3}x_3x_4 + m_{4,4}x_4x_4 + m_{4,5}x_4x_5 + \\ & m_{5,0}x_0x_5 + m_{5,1}x_1x_5 + m_{5,2}x_2x_5 + m_{5,3}x_3x_5 + m_{5,4}x_4x_5 + m_{5,5}x_5x_5 \end{aligned} \end{aligned} \tag{5.6}$$

Aunque la Ecuación 5.6 no es una matriz, es propiamente un polinomio, la hemos dejada expresada como se obtiene al multiplicar el vector $[x_0\ x_1\ x_2\ x_3\ x_4\ x_5]$ por el vector en 5.5, esto para observar que los términos del triángulo inferior son redundantes, ya que estos términos ya los tenemos en el triángulo superior[4]. Modificamos entonces la matriz M_{SQ} a una matriz triangular superior, formada como se muestra en la Ecuación 5.7.

$$M_{SQ} = \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} & m_{0,4} & m_{0,5} \\ 0 & m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} & m_{1,5} \\ 0 & 0 & m_{2,2} & m_{2,3} & m_{2,4} & m_{2,5} \\ 0 & 0 & 0 & m_{3,3} & m_{3,4} & m_{3,5} \\ 0 & 0 & 0 & 0 & m_{4,4} & m_{4,5} \\ 0 & 0 & 0 & 0 & 0 & m_{5,5} \end{bmatrix} \tag{5.7}$$

Finalmente y considerando esta nueva matriz M_{SQ}, los términos cuadráticos del polinomio, serán los mostrados en la Ecuación 5.8.

$$\begin{aligned} p_{SQ} = & \\ & \begin{aligned} & m_{0,0}x_0x_0 + m_{0,1}x_0x_1 + m_{0,2}x_0x_2 + m_{0,3}x_0x_3 + m_{0,4}x_0x_4 + m_{0,5}x_0x_5 + \\ & m_{1,1}x_1x_1 + m_{1,2}x_1x_2 + m_{1,3}x_1x_3 + m_{1,4}x_1x_4 + m_{1,5}x_1x_5 + \\ & m_{2,2}x_2x_2 + m_{2,3}x_2x_3 + m_{2,4}x_2x_4 + m_{2,5}x_2x_5 + \\ & m_{3,3}x_3x_3 + m_{3,4}x_3x_4 + m_{3,5}x_3x_5 + \\ & m_{4,4}x_4x_4 + m_{4,5}x_4x_5 + \\ & m_{5,5}x_5x_5 \end{aligned} \end{aligned} \tag{5.8}$$

Entonces, para generar la matriz M_{SQ}, inicializamos primeramente esta en ceros. Luego, de la lista original de términos cuadráticos (Ecuación 5.2) ver que variables son multiplicadas primero por x_0 (en este caso los términos son x_0x_1, x_0x_3, x_0x_5) y en el primer renglón de M_{SQ}, en la columna correspondiente al índice de dicha variable, colocar un uno (en este caso las variables involucradas son x_1, x_3, x_5 y por lo tanto los índices de las variables que representan a su

[4]Por ejemplo, el término $m_{1,0}x_0x_1$ de la primera columna, segundo renglón, es redundante con $m_{0,1}x_0x_1$ (primer renglón, segunda columna), ya que ambos involucran al término cuadrático x_0x_1. Basta entonces con considerar sólo a uno de estos términos, en caso que en el polinomio final deba aparecer dicho término.

vez las columnas del renglón uno serán $1, 3, 5$). Continuar ahora para x_1 y llenar el segundo renglón de M_{SQ}. Continuar así, hasta barrer todas las variables. En el Algoritmo 5 se muestra en resumen, como generar esta matriz. La matriz M_{SQ} para el ejemplo que nos ocupa, quedará finalmente como se muestra en la Ecuación 5.9.

$$M_{SQ} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.9}$$

Algoritmo 5 Algoritmo que genera la matriz para representar los términos cuadráticos de un polinomio

Entrada: Lista de productos cuadráticos de un polinomio ($x_i x_j$) y número de variables n, en estos polinomios

Salida: Matriz con elementos en $\mathbb{F}_2$ de la forma cuadrática de un polinomio

1: $M_{SQ} \leftarrow \mathbf{0}$ // Generar una matriz de orden $n \times n$ con todos sus elementos en cero
2: **while** Lista de productos cuadráticos $\neq \emptyset$ **do**
3: Sacar de la lista primer producto cuadrático $x_i x_j$
4: $M_{SQ_{i,j}} \leftarrow 1$
5: **end while**
6: **return** M_{SQ}

Conociendo entonces esta matriz, se pueden representar todos los términos cuadráticos del polinomio p. Para representar aún de forma más compacta esta matriz, se considera cada renglón que esta formado por $\{0, 1\}^n$, como un número binario y la representación entera de todos estos términos, será la conversión de estos números binarios a decimal. Entonces, los términos cuadráticos que se obtienen para la Ecuación 5.9 son:

$$\{21, 14, 10, 3, 2, 1\} \tag{5.10}$$

donde cada número como se dijo antes, es la representación en decimal del número tomado como binario de cada renglón.

2. Términos lineales. Para este caso, de la Ecuación 5.1, los términos lineales aparecen como se muestra en la Ecuación 5.11.

$$p_{Lin} = x_1 + x_3 + x_5 \tag{5.11}$$

La representación en enteros de los términos lineales, será la conversión a decimal del número binario que se obtiene al considerar un cero para una variable que no aparece en la Ecuación 5.11 y un uno para la variable que si aparece, empezando con la variable x_0 en la izquierda y terminando con la variable x_{n-1} a la derecha. Considerando el ejemplo que nos ocupa, el número binario a considerar es: 010101 que en decimal corresponde a:

$$\{21\} \tag{5.12}$$

3. Término constante. Este término y dado que estamos trabajando en $\mathbb{F}_2$ puede ser sólo 1 o bien no existir (0). La representación entera será entonces propiamente ese mismo valor, en este ejemplo es:

$$\{1\} \tag{5.13}$$

4. Representación entera completa: la representación de un polinomio (términos cuadráticos, lineales y constante), se formará por las n constantes que forman los términos cuadráticos (Ecuación 5.10), junto con la representación de los términos lineales (Ecuación 5.12) unido con la representación del término constante (Ecuación 5.13). Es decir, la representación de un polinomio en n variables se formará siempre de manera consistente de $n + 2$ constantes decimales. Para el ejemplo que nos ocupa, es:

$$\{21, 14, 10, 3, 2, 1, 21, 1\} \tag{5.14}$$

Para un sistema de m polinomios, se repite este proceso para cada polinomio y el resultado serán m conjuntos de $n + 2$ elementos cada uno.

Esta representación de formas cuadráticas, nos sirve entonces para representar de forma compacta un polinomio en n variables. Pero adicionalmente, podemos representar así cualquier matriz de orden $n \times n$, por ejemplo para representar la matriz M_s de la transformación afín o bien el vector $\mathbf{w}$ también de dicha transformación, ver Ecuación 3.1.

De la biblioteca de funciones `uov.sage`, la función `polySys2Int()`, que toma como parámetros `PolySyst, n, varName` tiene como objetivo, convertir un sistema de varios polinomios en `PolySyst`, formado de `n` variables y donde la variable es `varName`[5] a una representación entera. A su vez, la función `int2PolySys()`, con parámetros `IntPoly, n, x`, recibe un arreglo de números con la representación en enteros de varios polinomios (`IntPoly`) y los convierte a polinomios representados en su forma natural. En este caso, `x` es el vector de variables que tendrá el polinomio, es decir para $\mathcal{P}$ será $\mathbf{x}$ y para $\mathcal{P}'$, $\mathbf{a}$.

[5]La variable puede ser x para los polinomios $\mathcal{P}$ o bien a para los polinomios $\mathcal{P}'$

5.1.2 Biblioteca `uov.sage`

Como mencionamos en la sección anterior, las funciones `polySys2Int()` e `int2PolySys()` son las principales para la conversión de polinomios en su forma natural a representación entera y viceversa. Hay sin embargo más funciones en esta biblioteca, que están directamente relacionadas con el esquema AVNE. Las principales funciones, se describen a continuación.

1. - Función: `genPolyVectorUOV()`
   - Entradas:
     - `o`: número de variables aceite.
     - `v`: número de variables vinagre.
     - `x`: variable a usar en el polinomio (normalmente `a`, pero puede ser también `x`).
     - `Pi`: vector previamente definido para almacenar ahí cada uno de los polinomios generados.
     - `Debug = False`: bandera por default en falso; si se pasa como True, imprime el número del polinomio que se esta generando.
   - Salida: en `Pi` regresa un polinomio del tipo $\mathcal{P}'$, es decir con las características de los polinomios con `v` variables vinagre, `o` variables aceite y un total de `o` polinomios.
   - Uso: generar un conjunto de polinomios, con la principal característica de ser fácilmente invertibles, esto por su contrucción AVNE.
   - Ejemplo: `genPolyVectorUOV(o, v, a, PiUOV, Debug = True)`

2. - Función: `evalPoly()`
   - Entradas:
     - `Poly`: polinomio en n variables al que se desea evaluar.
     - x: variables que tiene el polinomio ($\mathbf{x}$ o $\mathbf{a}$).
     - `assig`: valores que deben tomar las variables.
   - Salida: regresa el polinomio original `Poly` evaluado en `assig`.
   - Uso: evalúa de forma eficiente, un polinomio `Poly` en la asignación `assig`, donde esta última, puede ser un valor constante o bien expresiones simbólicas. La evaluación busca hacerse de manera paralela, para que tome el menor tiempo posible.
   - Ejemplo: resultComputed = evalPoly(poly2Solve, x, xi)

3. - Funcion: `evalPolyVect()`
   - Entradas:
     - `PolyVect`: conjunto de polinomios a los que se desea evaluar.

- `x`: variable usada por cada uno de los polinomios, puede ser **x** o **a**.
- `assig`: valores a sustituir en cada uno de los polinomios.
- `y`: resultado de la evaluación del polinomio.
- `Debug = False`: bandera por default en falso, cuando se manda como True, imprime el número del polinomio que se esta evaluando.

- Salida: `y`, como un vector de la evaluación de cada polinomio en el vector de polinomios `PolyVect`.
- Uso: evaluar un conjunto de polinomios (llamado también vector de polinomios) en el valor dado en `assig`.
- Ejemplo: evalPolyVect(PiUOV, a, A, Pi, Debug = True)

4.
- Funcion: `S()`
- Entradas:
  - `Ms`: matriz generada aleatoriamente no-singular.
  - `x`: vector con los valores **x** a usar.
  - `vs`: vector aleatorio a usar en la transformación afín.
- Salida: ejecuta la operación $M_s \times x + vs$ y la regresa como valor de la función.
- Uso: cálculo de la transformación afín.
- Ejemplo: A = S(Ms, vector(x), vs)

5.
- Funcion: `Sinv()`
- Entradas:
  - `Ms`: matriz no-singular.
  - `S`: vector de dimensión n con el resultado de una transformación afín.
  - `vs`: vector de dimensión n.
- Salida: transformación inversa afín: $M_s.inverse() \times (S - v_s)$.
- Uso: encuentra la transformación inversa afín con base en la matriz `Ms`, a una transformación afín `S` y a un vector `vs`. Regresa esta transformación inversa, como valor de la función.
- Ejemplo: xi = Sinv(Ms, A, vs)

6.
- Funcion: `invertPiUOV()`
- Entradas:
  - `y`: imagen a satisfacer por un conjunto de polinomios, i.e. valores binarios que se desea cumplan los polinomios dados.
  - `o`: número de variables aceite.
  - `v`: número de variables vinagre.

- `x`: variables a usar (normalmente **a**).
- `PiUOV`: vector de polinomios a evaluar.

- Salida: imágen inversa de `y`: $\mathcal{P}' : x \mapsto y$
- Uso: regresa un vector A con los valores que deben tomar las variables **x** (normalmente las variables **a**) para que al sustituirlas en `PiUOV` entreguen un resultado igual al vector `y`.
- Ejemplo: A = invertPiUOV(y, o, v, a, PiUOV)

7.
- Funcion: `writeSK()`
- Entradas:
  - `Poly`: vector de polinomios que representan los polinomios secretos.
  - `Mat`: matriz no-singular.
  - `vec`: vector de la transformación afín.
  - `fileName`: nombre del archivo donde se escribirá la salida.
- Salida: dos archivos.
  - `filename.ir` con la representación en enteros de los polinomios de la clave secreta.
  - `filename.mv` con la representación en enteros de la matriz `Mat` y del vector `vec`.
- Uso: genera la representación en enteros de la clave privada completa, es decir de los polinomios secretos ($\mathcal{P}'$) y de la matriz y vector que conforman la transformación afín. Genera dos archivos con el nombre que se paso a la función (`filename`), pero con extensión `.ir` para los polinomios y extensión `.mv` para la matriz y vector. En el caso del archivo con extensión `.ir` guarda primero el número de variables de los polinomios, en el segundo renglón guarda el número de polinomios y después la representación entera de cada polinomio, tal como se explicó en la Sección 5.1.1. En relación al archivo con extensión `.mv`, guarda primero el número de variables, después la representación en enteros de la matriz y finalmente la representación en enteros del vector.
- Ejemplo: writeSK(PiUOV, Ms, vs, "SecretKey")

8.
- Funcion: `writePK()`
- Entradas:
  - `Pi`: polinomio público.
  - `fileName`: nombre del archivo a generar.
  - `n`: número de variables de los polinomios.
- Salida: dos archivos.

  - `filename` con el contenido en representación natural del vector de polinomios dado en `Pi`.
  - `filename.ir`, el cual contiene la representación en enteros de los polinomios en `Pi`.
- Uso: crea dos archivos, uno para guardar los polinomios públicos en su representación natural y otro con la representación en enteros.
- Ejemplo: writePK(Pi, "PublicKey", n)

9. - Funcion: `readInt()`
   - Entradas: `filename`: nombre del archivo a leer.
   - Salida:
     - `n`: número de variables en los polinomios.
     - `o`: número de polinomios.
     - `IntRep`: datos con la representación en enteros que contiene el archivo de entrada.
   - Uso: lee un archivo que contiene el número de variables que forman un vector de polinomios, el número total de polinomios y finalmente, la representación en enteros de cada polinomio.
   - Ejemplo: nn, oo, IntFile = readInt("SecretKey.ir")

10. - Funcion: `int2PolySys()`
    - Entradas:
      - `IntPoly`: una lista con la representación en enteros de un vector de polinomios.
      - `n`: número de variables del sistema de polinomios.
      - `x`: variables a usar en el anillo (**x** o **a**).
    - Salida: un vector de polinomios en el anillo de polinomios $\mathbb{K}[x]$ y variables **x** o **a**.
    - Uso: convertir una lista de polinomios en representación entera a su representación natural, asegurando además que queden en el anillo de polinomios de donde originalmente surgieron, en $\mathbb{K}[x]$ y en las variables **x** o bien **a**, según sea el parámetro que se pase.
    - Ejemplo: Poly = int2PolySys(IntFile, nn, a)

11. - Funcion: `readMat()`
    - Entradas: `filename`: nombre del archivo a leer.
    - Salida:
      - `n`: número de elementos en cada renglón y columna de la matriz leída, así como tamaño del vector a leído.

- `Ms`: matriz en su representación natural, es decir convertida de enteros a una matriz de orden $n \times n$.
- `vs`: vector con dimensión n, en su representación natural, convertido de lo leído del archivo de entrada.

- Uso: leer un archivo con la representación en enteros de una matriz de orden $n \times n$ y vector de dimensión n y convertirlos a su representación natural. Estos componentes, forman la transformación afín.
- Ejemplo: nn, mm, vv = readMat("SecretKey.mv")

5.2 Administración de claves

En la Tabla 5.2, se muestra la existencia del programa `genkey.sage`[6] que se usa para generar las claves privadas y públicas en un archivo cifrado. El programa `getkeys.sage` sirve para extraer del archivo cifrado generado por `genkey.sage` el conjunto de datos para la clave privada (polinomios privados, así como la matriz y vector de la transformación afín) y los polinomios para la clave pública. Por otra parte, el programa `hashCL.py` sirve para generar un dato resumen (*digesto*) del archivo con la clave pública, lo cual servirá para asegurar que quien tome dicha clave pueda confiar que la clave es la correcta. Presentamos a continuación los detalles de los puntos antes mencionados.

5.2.1 Generación de claves

Bajo el directorio `Keys` de la estructura mostrada en Programas, se encuentra el programa `genkey.sage` el cual tiene como objetivo, generar un sólo archivo que contiene la clave privada[7] y la clave pública[8], pero cifrado (usando AES-128) con la finalidad de no dejar en algún archivo la información anterior en texto plano, si no se tiene plena conciencia de lo que se está realizando. La sintaxis para correr este programa es la siguiente:

```
sage genkey.sage NumVars EncryptedFile
```

donde `NumVars` es un número que representa la cantidad de variables que formarán el sistema de polinomios[9] tanto para $\mathcal{P}$ como para $\mathcal{P}'$ y `EncryptedFile` es el nombre del archivo que se generará con las claves privada y públicas cifradas.

[6] Se muestra el archivo `genkey.sage.py` que es generado por Sage al correr el programa original `genkey.sage`. Este archivo se genera para cambiar algunos valores que permitan agilizar la ejecucón del programa, por ejemplo sustituir los valores enteros o reales por las definiciones propias de Sage. Esta mismo comentario aplica para el archivo `getkeys.sage.py` que se genera al correr el programa `genkeys.sage`

[7] Formada por la representación entera de los polinomios $\mathcal{P}'$, así como la representación entera de la matriz M_s y el vector vs que forman la transformación afín.

[8] Formada por la representación entera de los polinomios $\mathcal{P}$

[9] Recordar que el n es el número de variables y de este valor, se obtienen las variables aceite `o`, las variables vinagre `v` y el número de ecuaciones $m = o$, ver detalles en la Sección 3.3.1

Este programa, primeramente valida que se le pasen los parámetros antes mencionados y con esto define la cantidad de variables aceite, vinagre y el número de ecuaciones de cada sistema de polinomios. Posteriormente, define las variables **x** y **a** que se usarán por los polinomios $\mathcal{P}$ y $\mathcal{P}'$ así como el anillo de polinomios a usarse. Con esta información, define propiamente los vectores que contendrán los valores a los que deben evaluar los polinomios $\mathcal{P}$ y $\mathcal{P}'$, en este caso y y $yUOV$, la matriz y vector de la transformación afín (M_s y v_s) y finalmente los vectores que contendrán los m polinomios públicos $\mathcal{P}$ así como los m polinomios privados $\mathcal{P}'$.

Con las definiciones anteriores realizadas, y de la biblioteca `uov.sage` el programa usa la función `genRndMatrix(Ms)` para generar la matriz M_s de la transformación afín y la función `genRndVector(vs)` para generar el vector vs de la misma transformación afín. Ahora, usando la función `genPolyVectorUOV(o, v, a, PiUOV)` genera aleatoriamente el sistema de polinomios $\mathcal{P}'$ con o variables aceite, v variables vinagre, representando a las variables como a_i y el sistema de ecuaciones quedará en el vector $PiUOV$. Usa ahora la transformación afín, para obtener expresiones de las variables a_i en función de las variables x_i. Para esto ejecuta la función `S(Ms, vector(x), vs)` creando la variable A que contiene los valores de cada a_i en función de x_i. Con estos valores, genera el vector de polinomios públicos Pi, usando la función `evalPolyVect(PiUOV, a, A, Pi)`.

Empleando la función `readPP` de la biblioteca `cryptokeys.sage`, el programa pide una frase, que se usará para cifrar/descifrar la clave privada. Es importante aclarar, que la clave pública aunque se cifra usando AES-128, no usa esta frase y esta clave pública se podrá descifrar sin necesidad de proporcionar frase alguna, como se verá en la Sección 5.2.2. Con esta información y usando la misma biblioteca de funciones `cryptokeys.sage`, invoca ahora a la siguiente función, `writeKeysEnc(PiUOV, Ms, vs, Pi, pp, easyK, encFile)` que usando el sistema de polinomios privados $\mathcal{P}'$, la matriz y vector de la transformación afín (M_s y v_s), el sistema de polinomios públicos $\mathcal{P}$, la frase mencionada antes (pp) y una frase simple que esta incrustada en el código de este programa para cifrar los polinomios públicos[10], cifra entonces la clave privada[11], luego cifra con la frase simple la clave pública y concatena ambos cifrados, para escribirlos finalmente al archivo cuyo nombre se proporcionó como parámetro al programa en `encFile`.

5.2.2 Extracción de claves

Una vez que se ha generado un archivo cifrado que contiene la clave privada y la clave pública, veremos ahora como con el programa `getkeys.sage` se pueden

[10]Dado que los polinomios públicos no tienen porque ocultarse, dado que precisamente pueden ser conocidos por todos, es que se usa una clave para cifrar estos polinomios que esta incrustada en el programa `genkey.sage`. La razón del cifrado, es sólo para que el archivo que se genera sea homogéneo, pero la clave pública se podrá descifrar sin proporcionar frase alguna, como veremos en la Sección 5.2.2

[11]Usa el resultado de aplicar una función picadillo, en este caso SHA256, a la frase, para tener un tamaño constante como clave de cifrado para AES-128

recuperar las claves antes mencionadas. Recordar que el programa `getkeys.sage` se encuentra en el directorio `Keys` del enlace `Programas` (ver Tabla 5.2). La sintaxis para correr este programa es la siguiente:

```
getkeys [-S] EncryptedFile [SKFile] PKFile
```

donde los parámetros entre corchetes son opcionales, pero simultáneos, es decir o aparecen los dos o no se proporcionan los dos. Entonces, cuando se dan estos parámetros opcionales el comando quedaría como:

```
getkeys -S EncryptedFile SKFile PKFile
```

Esta forma del comando, es para descifrar el archivo `EncryptedFile` extrayendo de él tanto la clave privada como la pública. La clave privada quedará en dos archivos, el primero `SKFile.ir` que contendrá la representación en enteros de los polinomios privados ($\mathcal{P}'$) y el archivo `SKFile.mv` que contendrá la representación en enteros de la matriz y vector (M_s, v_s) de la transformación afín. Al mismo tiempo, se generarán dos archivos más para la clave pública, el primero llamado `PKFile` (sin extensión adicional) y que contendrá los polinomios de la clave pública en su forma natural; se generará además el archivo `PKFile.ir` con la representación en enteros de los polinomios de la clave pública. Para esta forma del comando, se solicitará de forma interactiva la frase usada para cifrar el conjunto de elementos de la clave privada. Si esta no se proporciona correctamente, el programa no descifra esta clave y continua generando solamente el descifrado de la clave pública.

La segunda forma del comando es la siguiente:

```
getkeys EncryptedFile PKFile
```

Con este formato, sólo se descifrará y generará la clave pública, sin pedir para esta clave, (como en el caso anterior), alguna frase de descifrado. Igual que antes, se generarán dos archivos `PKFile` y `PKFile.ir` con los polinomios de la clave pública en su forma natural y con la representación en enteros de los polinomios de la clave pública respectivamente.

Las funciones empleadas por el programa `getkeys.sage`, se encuentran en la biblioteca `cryptokeys.sage`. El programa `getkeys.sage` realiza primero las validaciones necesarias, para asegurar que se está usando la sintaxis correcta. Después, llama a la función `getCTofSK_PK(encFile)`, pasando a la función el nombre del archivo que contiene las claves privadas y públicas cifradas. Esta función extrae primeramente el texto cifrado que corresponde a la clave privada y después extrae el texto cifrado de la clave pública, esto del archivo que se dió a la función. Si no se pueden extraer correctamente estos textos cifrados, regresa un error y entonces el programa termina.

Si se llamó al programa pidiendo que se extraiga la clave privada, entonces el programa llama a la función `askPP` para que pida la frase que se usó para cifrar los elementos de la clave privada. Después, llama a la función `getSK(CT_SK, SKFile + extPI, SKFile + extMV, pp)` y le pasa el texto cifrado correspondiente a la clave privada, el nombre del archivo donde se almacenará la representación entera de los

polinomios privados, el nombre del archivo donde se almancenará la representación entera de la matriz y vector de la transformación afín y finalmente la frase que se debe usar para descifrar. Si pudo descifrar, genera los archivos con la información de la clave privada y continua ahora con la pública. La función regresa un error si no pudo descifrar y continua también con el descifrado de la clave pública.

Para el descifrado de la clave pública, el programa llama a la función `getPK(CT_PK, PKFile + extPI, PKFile, easyK)`, a la que le pasa el texto cifrado de la clave pública, el nombre del archivo donde almacenará la clave pública representada como enteros, el nombre del archivo donde almacenará los polinomios de la clave pública expresados de forma natural y una clave de descifrado que esta incrustada en la biblioteca `cryptokeys.sage`. Después de esto, el programa termina y los archivos han sido generados.

5.2.3 Generación resumen clave pública

Una vez que se tiene el archivo con la clave pública en su representación de enteros, se debe hacer público para poder implementar el protocolo de autenticación o bien el de cifrado. Con la finalidad de asegurar que el archivo original es el que está llegando al destinatario, se deberá generar por medio de una función picadillo un resumen (digesto) del archivo a transferir. Después, el receptor de la clave pública deberá generar el resumen del archivo recibido y compararlo con el originalmente generado, el cual deberá estar al alcance del receptor de algún sitio seguro en internet. Este procedimiento no se trata en este trabajo y solamente se realizó el programa que se describe a continuación, para generar el resumen de la clave pública. El programa se llama `hashCL.py` y se ejecuta de la siguiente manera:

```
python hashCL.py File2Hash HashedFile
```

El parámetro `File2Hash` es el nombre del archivo que contiene la clave pública y `HashedFile` es el nombre del archivo que almacenará el resumen de la clave pública. Este programa realiza lo siguiente: primeramente, valida que se proporcionen los datos de los dos archivos necesarios para este programa, si no es el caso, el programa termina con una condición de error. Si se proporcionaron los nombre adecuados y existe el archivo con la clave pública y se pudo crear el archivo donde se guardará el resumen, entonces genera dicho resumen y lo guarda en el archivo correspondiente.

5.2.4 Codificación en ASN.1

Con la finalidad de poder transferir archivos con datos, entre diversas plataformas, se ha utilizado normalmente en el ambiente de seguridad (y en muchos más), el convertir los archivos a transferir en alguna codificación en ASN.1. En (Dubuisson, 2000) se trata a profundidad el tema de ASN.1 y éste sale del alcance de este trabajo, por lo que sólo explicaremos la forma en que se generan los archivos codificados en DER (Distinguished Encoding Rules) y la decodificación a los archivos originales.

Los programas que desarrollamos para codificar/decodificar en ASN.1 los archivos con la representarción entera de los polinomios públicos se desarrollaron en C y utilizamos una biblioteca para ASN.1 desarrollada por GNU, llamada *libtasn1*. La versión usada de dicha biblioteca es la 4.3 y el desarrollo se realizó en un ambiente Linux.

El archivo que tiene la representación en enteros de la clave pública deberá transferirse por algún método a la entidad que lo vaya a usar para establecer el protocolo de autenticación o bien de cifrado explicados en el Capítulo 4. Sin embargo no podemos estar seguros como interpretará estos enteros el receptor ni tampoco la forma exacta en que éstos se transferirán. Debido a ello es conveniente usar ASN.1 que como una de sus funciones primeras es la de describir los datos a manejar y almacenarlos en un formato (en nuestro caso codificados como DER) que no dependa de la plataforma que se use y de esta manera, transferirlo y en el destino decodificarlo al formato adecuado para que lo maneje apropiadamente ese destinatario. Para ello, es necesario primero realizar un archivo que describa los datos a manejar. Este archivo se llama `pkInt.asn` y se encuentra en el directorio `/ASN.1/encoder/development` del enlace `Programas`. En ese mismo directorio, se encuentra el programa fuente que desarrollamos para realizar la codificación DER de un archivo con la representación entera de una clave pública. El archivo con el programa fuente, se llama `encode.c` y esta desarrollado en C. Para generar el programa ejecutable, se deben seguir los siguientes pasos:

1. Primeramente, se deberán generar las estructuras de datos requeridas por ASN.1 con base en el archivo que describe los datos en el archivo que contiene la clave pública. Como mencionamos antes, este archivo tiene el nombre de `pkInt.asn` y para generar estas estructuras, se debe ejecutar el comando siguiente[12]

   ```
   asn1Parser pkInt.asn
   ```

   El comando anterior, generará las estructuras necesarias en el programa `pkInt_asn1_tab.c`, mismo que se deberá encadenar con el programa fuente que se cree para realizar propiamente la codificación (ver punto 2).

2. Una vez creado el programa con las estructuras de datos para generar en C la codificación DER, es necesario compilar el programa que realizamos para dicha codificación. Esto se realiza ejecutando el siguiente comando:

   ```
   gcc -o encode encode.c pkInt_asn1_tab.c ‘pkg-config libtasn1 --libs‘
   ```

   Se generará un programa ejecutable para un ambiente Linux, llamado `encode` que como veremos a continuación sirve para codificar un archivo con la clave pública en representación entera

[12] Recordar que para que este comando se pueda ejecutar, debe estar instalada la biblioteca *libtasn1*, versión 4.3

3. Para generar el archivo ejecutable que decodifica un archivo DER-ASN.1, se deben seguir pasos similares, pero en este caso, el directorio de trabajo será `/ASN.1/decoder/development`. El comando para generar las estructuras de datos es igual al del punto 1 y para generar el programa ejecutable del decodificador, hay que ejecutar:

```
gcc -o decode decode.c pkInt_asn1_tab.c ‘pkg-config libtasn1 --libs‘
```

Para mayor facilidad, se copiaron los dos archivos ejecutables (`encode` y `decode`) directamente al directorio `/ASN.1`. Entonces, para realizar la codificación a DER de una clave pública en representación entera, se deberá ejecutar el siguiente comando:

```
ASN.1/encode PublicKeyIntFile ASN1BinaryFile
```

Es decir, se pasan como parámetros al programa de codificación DER de ASN.1 el nombre del programa que contiene la representación en enteros de la clave pública y el archivo destino donde quedará dicha codificación.

Para el caso de decodificar un archivo en DER y regresarlo a su formato original que es una representación en enteros de una clave pública, el comando a ejecutar es:

```
ASN.1/decode ASN1BinaryFile PublicKeyIntFile
```

Y como se observa de este comando, al programa `decode` se le pasan como parámetros, el nombre del archivo que contiene la codificación DER y el nombre del archivo que se creará con la representación entera de la clave pública.

5.2.5 Codificación Base64

Una vez que se tiene el archivo ASN.1, que representa a la clave pública en su representación en enteros, como se explicó en la Sección 5.2.4, hay que convertir ese archivo a una codificación Base 64. El motivo de esto, es que el archivo ASN.1 contiene caracteres no desplegables, por el tipo de codificación que realiza ASN.1 y muchas veces, si se desea enviar directamente como cuerpo de un correo el contenido de la clave, los caracteres no desplegables representarían un problema en esta transmisión, ya que probablemente no llegarían de manera fiel al destinatario. Por esto, se creo la codificación Base 64, la cual invariablemente a cualquier archivo lo representa como una serie de caracteres ASCII desplegables, como son las letras mayúsculas, minúsculas, los diez dígitos y los caracteres + y /.

Bajo el directorio `Base64` del enlace `Programas`, se encuentran los programas `ebase64.c` y `ebase64` (programa fuente y ejecutable) que codifican un archivo cualquiera a Base 64 y `dbase64.c` y `dbase64` (también fuente y ejecutable) que decodifican un archivo Base 64 regresándolo al tipo original.

Para codificar un archivo a Base 64, se debe ejecutar el siguiente comando:

```
./ebase64 BinaryFile Base64File [‘‘algún texto’’]
```

En este caso, `BinaryFile` es el nombre del archivo que se desea codificar (normalmente de algún tipo binario, como los producidos por ASN.1, aunque esto no es restricción), `Base64File` es el nombre del archivo que se creará con la codificación Base 64 y opcionalmente se puede proporcionar un texto que aparecerá al inicio y final del archivo con la codificación que se genere.

Para el caso de la decodificación de una archivo Base 64 que deberá generar el archivo original, se debe ejecutar el siguiente comando:

```
./dbase64 EncodedFile DecodedFile
```

Donde `EncodedFile` es el nombre del archivo que contiene información en Base 64 y `DecodedFile` es el nombre del archivo que se generará con la decodificación de Base 64.

5.2.6 Ejemplo del ciclo completo

Hemos mencionado en las secciones anteriores como correr cada uno de los procesos que intervienen en la generación y manejo de claves. Veremos ahora como se integran cada una de esas partes, para plantear un esquema que abarque el ciclo completo, es decir, desde que se generan las claves, hasta que se genera la clave pública del lado de la entidad pública.

En el directorio principal del enlace `Programas`, se encuentran dos programas para correrse bajo la linea de comandos del interpretador de Linux, Bash. El primer programa es `genKeys.sh` y tiene como finalidad generar las claves privada y pública y realizar la codificación de la clave pública para ponerla disponible. Este programa, sólo debe estar disponible para la entidad privada (por ejemplo un *probador*) que es quien deberá tener acceso a la clave privada.

El segundo programa, es `decKey.sh` y sirve para decodificar una clave pública, así como para generar el resumen (digesto) de esta clave y ver si éste es igual al que también se ha publicado. Este programa lo debe correr una entidad pública (por ejemplo un *verificador*) ya que sólo tiene acceso a dicha clave pública y no puede hacer nada dañino con esta clave.

Veamos entonces como funciona cada programa.

Generación completa de claves

Como mencionamos antes, el programa `genKeys.sh` corre directamente en la línea de comandos en una sesión Bash de un equipo con Linux. Este programa se diseño para requerir el mínimo de parámetros, por lo que sólo hay que proporcionarle el número de variables con que se deberá crear cada polinomio. La forma de ejecutarse es la siguiente:

```
./genKeys.sh NoVars 2>>/tmp/errors
```

Donde `NoVars` es un número entero, que representa el número de variables con que se crearán los polinomios de la clave privada y pública y `2>>/tmp/errors` es para que los posibles mensajes de error que mande algún programa no se mezclen en la pantalla con la salida natural de dichos programas, sino más bien queden registrados en el archivo que se proporciona, en este caso `/tmp/errors`. Los procesos que ejecuta este programa son los siguientes:

1. Valida que se le proporcione el parámetro del número de variables que contendrá cada polinomio tanto de la clave pública como de la privada. Si este parámetro no se proporciona, el programa termina con un mensaje de error.

2. Si el parámetro anterior es proporcionado, y considerando que este programa utilizará los programas que se han venido explicando en toda la Sección 5.2, entonces se revisa que la estructura de directorios mostrada en la Tabla 5.1 exista a nivel de donde esta el programa `genKeys.sh`. Si esto no es correcto el programa termina.

3. Genera ahora las claves privadas y públicas en un archivo cifrado, el cual tomará el nombre de `Keys.enc`. Este archivo y todos los que se generen por el programa `genKeys.sh` quedarán al final bajo el directorio `files`.

4. Descifra ahora el archivo con las claves cifradas. Los archivos generados tendrán la forma `SK+NoVars` y `PK+NoVars` con las extensiones `.ir` para los archivos con la representación de enteros de los polinomios, la extensión `.mv` para los archivos con la codificación en enteros de la matriz y vector de la transformación afín o sin extensión cuando se trate de la clave pública mostrando los polinomios en su forma natural. En este caso, `NoVars` es el número que se proporcionó al programa y que representa el número de variables de los polinomios.

5. Genera el resumen (digesto) del archivo con la representación en enteros de la clave pública y lo deja en el archivo `PK+NoVars` con extensión `.hash`.

6. Finalmente, genera el archivo con la codificación para ASN.1 (en este caso codificación DER) y el archivo codificado en Base 64. Ambos son para la clave pública por lo que el archivo se llamará `PK+NoVars` con extensión `.asn1` y `.B64` respectivamente.

En la Figura 5.2 tenemos el diagrama de flujo completo de este proceso. La flecha que se encuentra en la parte inferior derecha de este diagrama significa que tanto el archivo con el resumen de la clave pública, como el archivo con la clave pública codificada en Base 64, se dejan disponibles de alguna manera en una red, ya sea privada o pública y los detalles de esto, no se tratan en esta tésis.

Veamos la forma en que se interactua con este programa. La Figura 5.3 muestra que el directorio donde se correra el programa `genKeys.sh` cuenta con la estructura de directorios adecuada y que el directorio `files` esta vacío. Se muestra también el

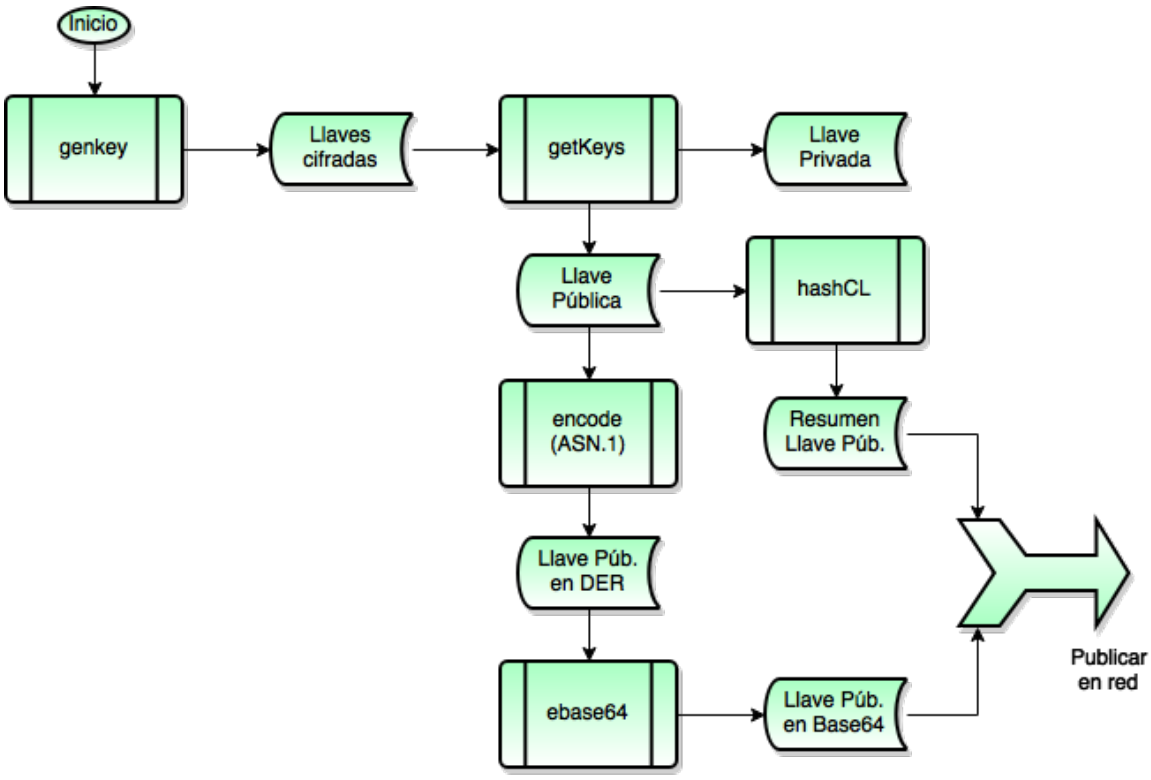

Figura 5.2: Diagrama de flujo entidad privada

comando para correr este programa, a punto de lanzarse, en este caso para generar polinomios con doce variables que corresponderá a un conjunto de 4 polinomios[13]

Una vez lanzado el comando de ejecución del programa `genKeys.sh`, se corre propiamente el programa `genkey.sage` y como se explicó en la Sección 5.2.1 pide ahora la frase para cifrar la clave privada en el archivo cifrado que se genera, en este caso en el archivo llamado `Keys.enc`. La Figura 5.4 muestra esta parte.

Después de esto, se correrá de forma automática, el programa `getkeys.sage` y como se puede apreciar en la Figura 5.5 pide la frase para descifrar el archivo con la clave privada, es decir la frase que se acaba de dar como se mencionó en las lineas anteriores. Si la frase es correcta, el programa continua descifrando la clave privada y luego la clave pública (para la cual no se pide frase). Continua ahora corriendo el programa `hashCL.py` para generar un resumen de la clave pública y terminar generando el archivo codificado en DER de ASN.1 y el archivo Base 64 que es el que se publicará.

Revisando el contenido del directorio `files` (ver Figura 5.6), se generaron los archivos que se muestran en la Tabla 5.3, donde se incluye también que contiene cada archivo.

[13]Recordar como vimos en 3.3, que el número de variables debe ser igual al número de variables aceite o mas las variables vinagre v y que las variables $v = 2o$ i.e. $n = 3o$ y $o = m$ donde m es el número de ecuaciones.

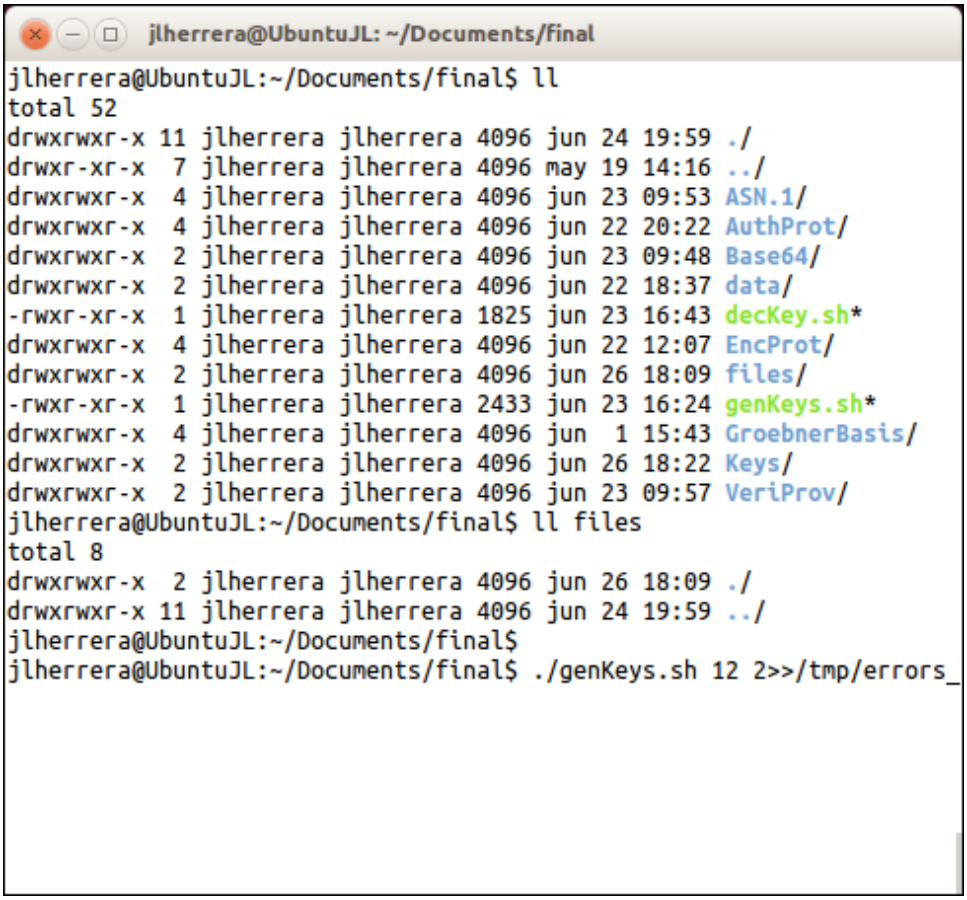

Figura 5.3: Condiciones iniciales para correr el programa `genKeys.sh`

En la Figura 5.6 se muestra también el contenido del archivo `PK12.B64` que contiene la codificación en Base 64 del archivo binario `PK12.asn1`. Se puede apreciar que este archivo es completamente desplegable, ya que sólo contiene caracteres alfanuméricos (y posiblemente los símbolos + y /). Así mismo, aparece la leyenda `-----BEGIN MQ Polynomials-----` y `-----END MQ Polynomials-----` esto debido a que al invocar al programa de codificación Base 64, se paso el parámetro opcional de la leyenda que se desea incluir en este archivo, en este caso se paso la cadena `"MQ Polynomials"`. En esta misma figura, se muestra el contenido del archivo `PK12.ir` es decir la representación en enteros de los polinomios de la clave pública. Como se puede apreciar y de acuerdo a la teoría expuesta en la Sección 5.1.1 en el primer renglón aparece el número de variables, en este caso `12`, en el siguiente renglón, el número de polinomios (`4`) y después cuatro renglones más (uno por cada polinomio) con *12+2=14* números enteros, los primeros doce representando los términos cuadráticos de un polinomio, el siguiente número representando los términos lineales y el último número representando el término constante del mismo polinomio.

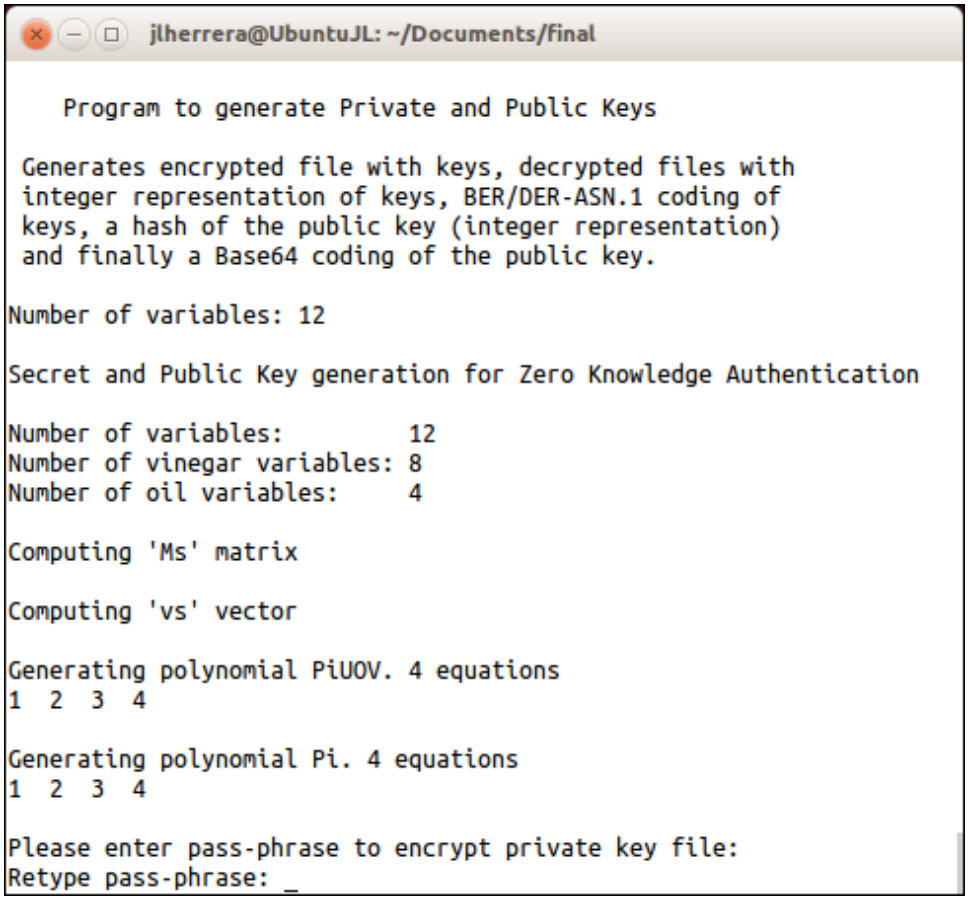

Figura 5.4: `genKeys.sh` pidiendo la frase para cifrar el archivo de claves.

Decodificación completa de clave pública

Una entidad pública, que en el caso del protocolo de autenticación se representa por el *verificador* o en el caso del protocolo de cifrado es el personaje `Beto`, recibe solamente la clave pública en su representación Base 64 y el resumen (digesto) de dicha clave. El programa `decKey.sh` que también se encuentra en el directorio raíz del enlace `Programas`, tiene como finalidad decodificar el archivo antes mencionado a su origen que es un archivo codificado en DER de ASN.1, para después generar la clave pública en su representación de enteros y de ahí, generar su resumen (digesto). Este resumen deberá compararse con el resumen recibido y sólo en caso que sean iguales, se podrá aceptar como válida la clave pública que se ha generado. El programa `decKey.sh` debe ejecutarse de la siguiente manera:

```
./decKey.sh FileNameB64 hashFileName
```

Donde `FileNameB64` es el nombre del archivo que debe estar en el directorio `files` y cuyo contenido debe ser la codificación Base 64 del archivo ASN.1 de la clave pública en su representación en enteros. Por otra parte, `hashFileName` es el nombre del archivo que también se debe encontrar en el directorio `files` y cuyo contenido

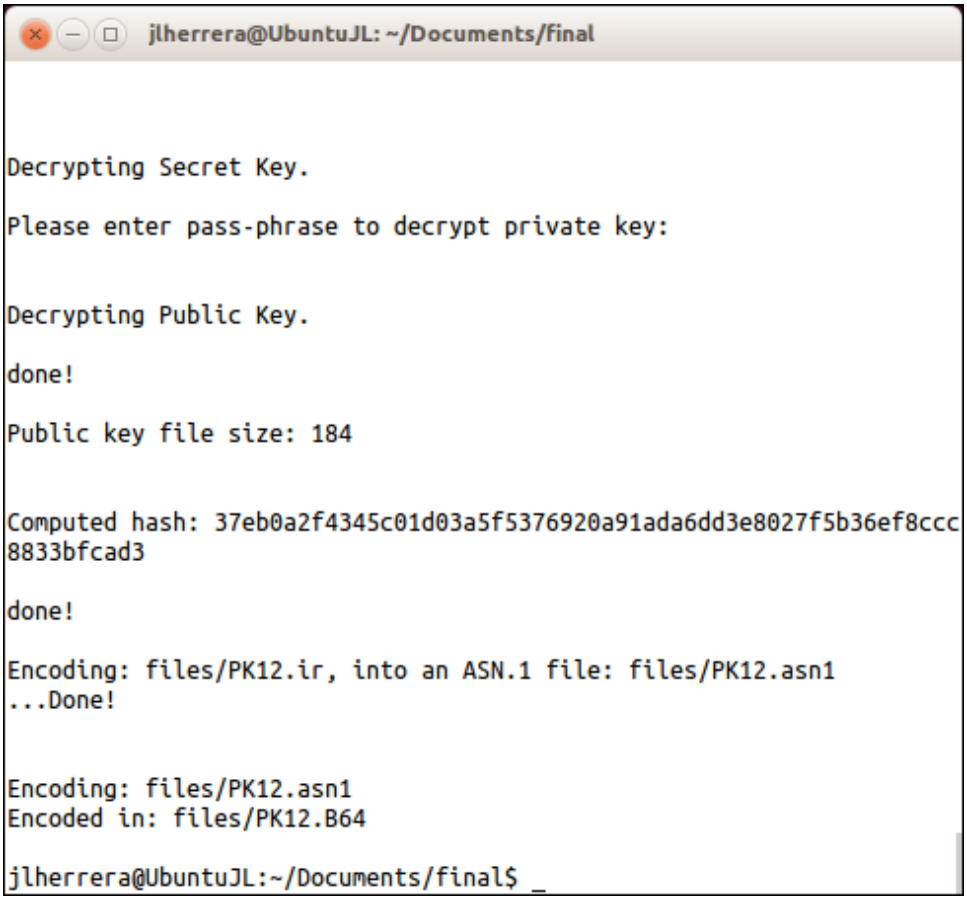

Figura 5.5: `genKeys.sh` pidiendo la frase para descifrar el archivo de claves y realizando las codificaciones finales.

debe ser el resumen de la clave pública original en su representación en enteros. Los pasos que ejecuta este programa son los siguientes:

1. Primero valida que se le hayan proporcionado los dos nombres de archivo: `FileNameB64` y `hashFileName`. Si no es el caso, o bien si estos archivo no existen bajo el directorio `files` el programa manda un mensaje de error y termina.

2. Se valida ahora que la estructura de directorios de los programas en `Programas` sea la correcta

3. Considerando entonces, que los archivos antes mencionados tiene el primero la codificación Base 64 de la codificación ASN.1 de la clave pública en su representación en enteros y que el segundo es el archivo con el resumen (digesto) de la clave pública en su representación en enteros, el programa realiza la conversión del archivo `FileNameB64` a codificación DER en ASN.1, produciendo el archivo `FileNameB64.asn1`

4. Toma ahora el archivo `FileNameB64.asn1` y lo decodifica a la clave pública en

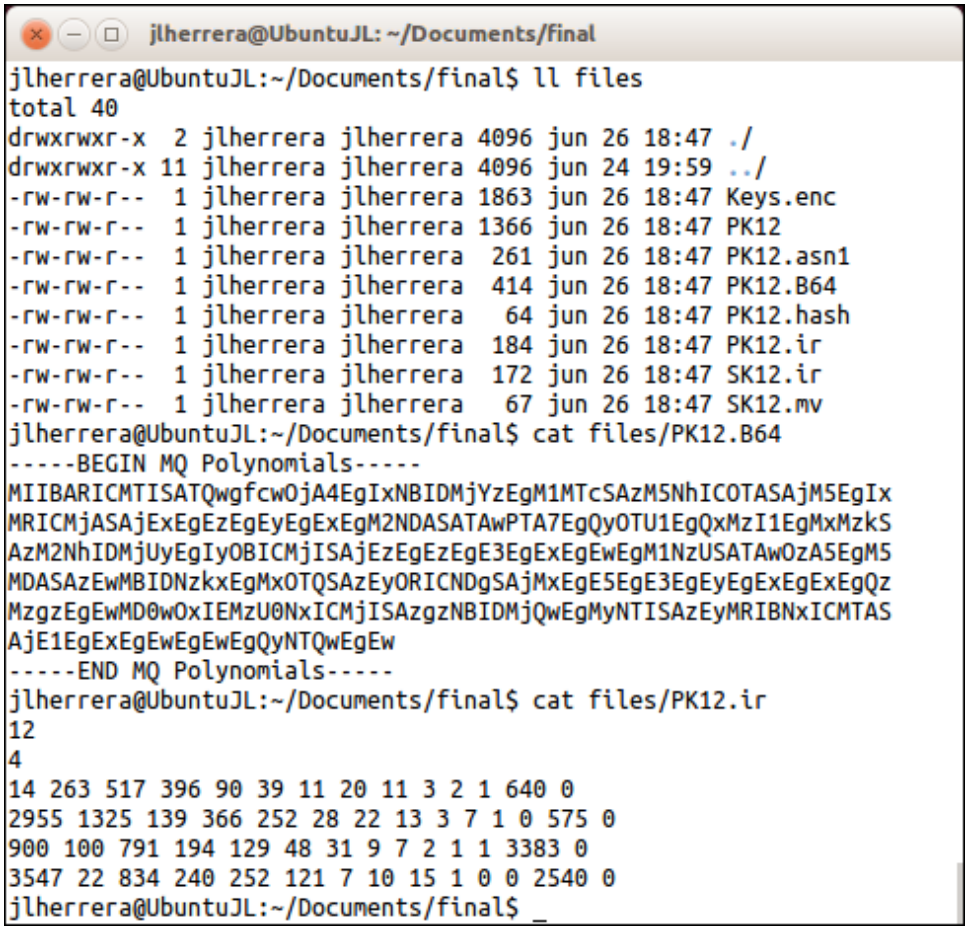

Figura 5.6: Archivos generados por `genKeys.sh` y contenido de la clave pública en Base 64 y en su representación en enteros.

su representación de enteros, generando el archivo `FileNameB64.ir`

5. Genera el archivo resumen (digesto) del archivo que se generó con la clave pública (`FileNameB64.ir`) dejándolo en `FileNameB64.hash` y finalmente lo compara con el archivo resumen (digesto) recibido que se paso a este programa en `hashFileName`. Si son idénticos, manda un mensaje que la clave pública producida es válida y si no son iguales, el mensaje es de NO usar este archivo.

En la Figura 5.7 se muestra el diagrama de flujo de los pasos antes mencionados. La flecha que se encuentra en la parte media izquierda, indica que de la red (ya sea por internet o una red local) se hacen del conocimiento de esta entidad pública, el archivo resumen (digesto) *correcto* de la verdadera clave pública en su representación en enteros y un archivo codificado en Base 64, que contiene la codificación DER-ASN.1 de la clave pública en su representación en enteros, teóricamente sin ninguna alteración.

Veamos como es la interacción con el programa `decKey.sh`. En este caso, tomaremos los archivos que se han generado del ejemplo con `genKeys.sh`. De la Tabla 5.3 se

Archivo	Descripción
`Keys.enc`	Archivo cifrado en AES-128, que contiene la clave privada y la clave pública
`PK12`	Contiene los polinomios de la clave pública mostrados en su forma natural
`PK12.asn1`	Codificación DER de ASN.1, del archivo PK12.ir
`PK12.B64`	Codificación en Base 64 del archivo PK12.asn1
`PK12.hash`	Resumen (digesto) del archivo PK12.ir
`PK12.ir`	Polinomios de la clave pública en su representación en enteros
`SK12.ir`	Polinomios de la clave privada, en su representación en enteros
`SK12.mv`	Matriz y vector de la transformación afín (parte de la clave secreta) en su representación en enteros

Tabla 5.3: Contenido del directorio `files` después de correr el programa `genKeys.sh`

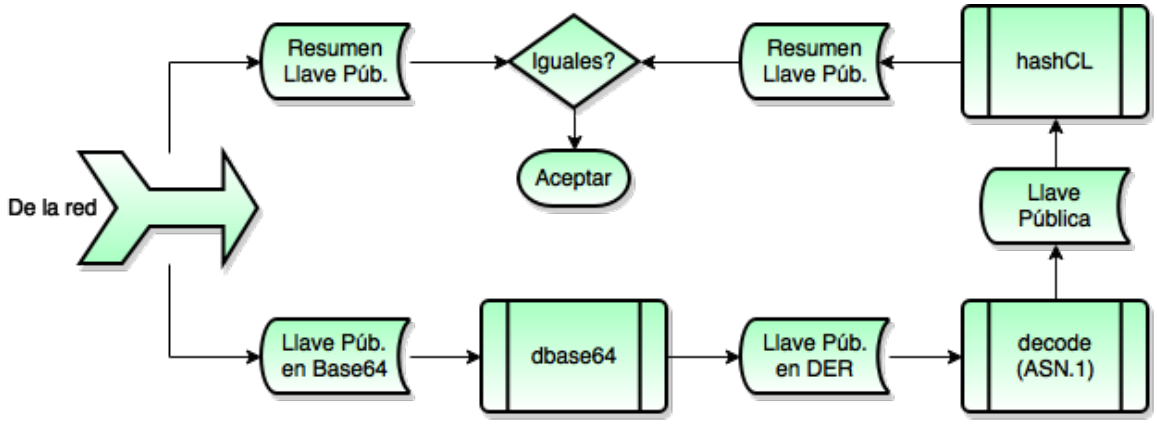

Figura 5.7: Diagrama de flujo entidad pública

usarán los archivos `PK12.B64` que contiene la codificación Base 64 de la clave pública previamente codificada en DER-ASN.1 y el archivo `PK12.hash` que contiene el resumen (digesto) de la clave pública original en su representación de enteros. El comando a ejecutarse es[14]

```
./decKey.sh PK12.B64 PK12.hash
```

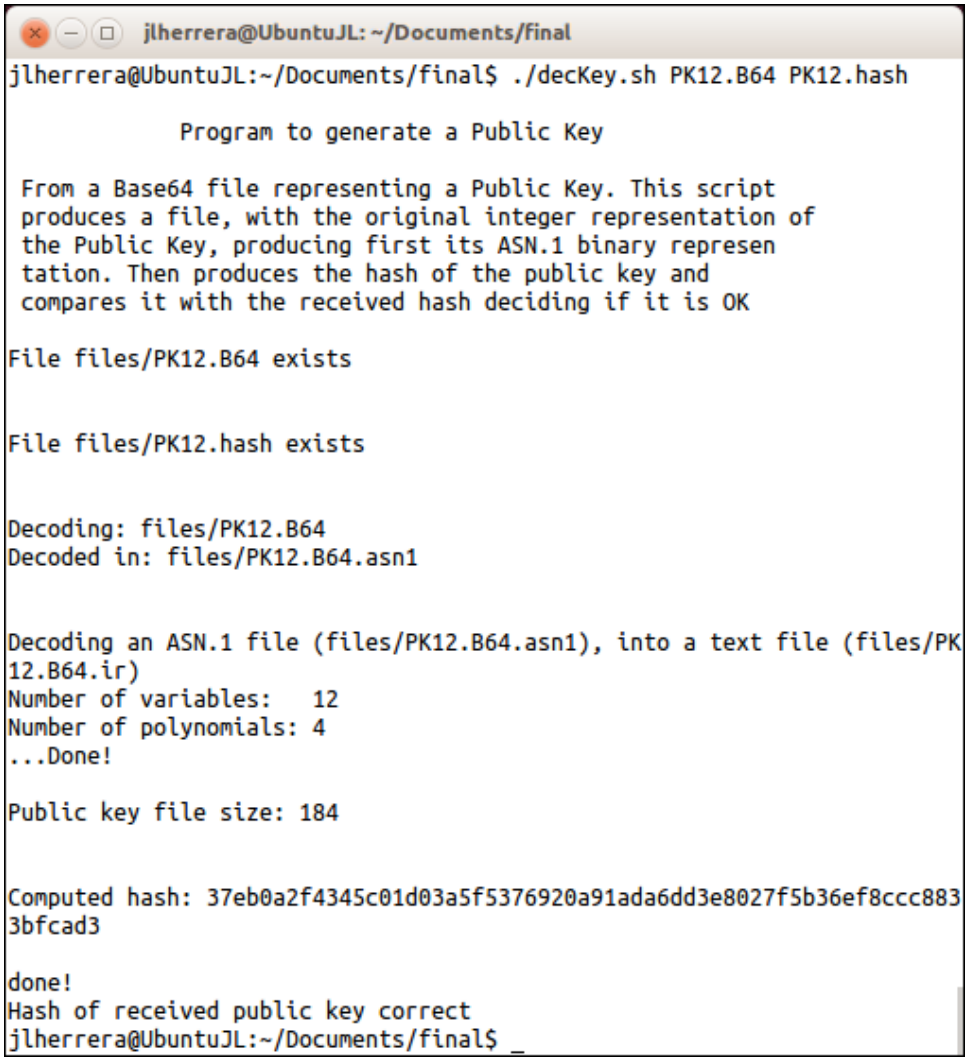

Figura 5.8: Ejecución del programa `decKey.sh` con un resultado exitoso

La Figura 5.8 muestra la salida del comando anterior, que resulta en una decodificación de la clave pública exitosa, dado que el archivo resumen (digesto) recibido (`PK12.hash`) es igual al generado por este proceso (`PK12.B64.hash`). La Figura 5.9, muestra el contenido con el que queda el directorio `files` después de la ejecución

[14]Recordar que los archivos `PK12.B64` y `PK12.hash` deben existir bajo el directorio `files` como sucede en este caso.

exitosa del comando `decKey.sh PK12.B64 PK12.hash`. Los archivos generados son: `PK12.B64.asn1` con la codificación DER-ASN.1 que se obtiene al procesar el archivo `PK12.B64`; el archivo `PK12.B64.ir` con la representación en enteros de la clave pública (cuyo contenido se muestra al final de la misma Figura 5.9) y el archivo `PK12.B64.hash` que corresponde al resumen (digesto) del archivo `PK12.B64.ir`.

```
jlherrera@UbuntuJL: ~/Documents/final
jlherrera@UbuntuJL:~/Documents/final$ ll files
total 52
drwxrwxr-x  2 jlherrera jlherrera 4096 jun 27 23:08 ./
drwxrwxr-x 11 jlherrera jlherrera 4096 jun 27 22:57 ../
-rw-rw-r--  1 jlherrera jlherrera 1863 jun 26 18:47 Keys.enc
-rw-rw-r--  1 jlherrera jlherrera 1366 jun 26 18:47 PK12
-rw-rw-r--  1 jlherrera jlherrera  261 jun 26 18:47 PK12.asn1
-rw-rw-r--  1 jlherrera jlherrera  414 jun 26 18:47 PK12.B64
-rw-rw-r--  1 jlherrera jlherrera  261 jun 27 23:07 PK12.B64.asn1
-rw-rw-r--  1 jlherrera jlherrera   64 jun 27 23:07 PK12.B64.hash
-rw-rw-r--  1 jlherrera jlherrera  184 jun 27 23:07 PK12.B64.ir
-rw-rw-r--  1 jlherrera jlherrera   64 jun 26 18:47 PK12.hash
-rw-rw-r--  1 jlherrera jlherrera  184 jun 26 18:47 PK12.ir
-rw-rw-r--  1 jlherrera jlherrera  172 jun 26 18:47 SK12.ir
-rw-rw-r--  1 jlherrera jlherrera   67 jun 26 18:47 SK12.mv
jlherrera@UbuntuJL:~/Documents/final$ more files/PK12.B64.ir
12
4
14 263 517 396 90 39 11 20 11 3 2 1 640 0
2955 1325 139 366 252 28 22 13 3 7 1 0 575 0
900 100 791 194 129 48 31 9 7 2 1 1 3383 0
3547 22 834 240 252 121 7 10 15 1 0 0 2540 0
jlherrera@UbuntuJL:~/Documents/final$ _
```

Figura 5.9: Contenido del directorio `files`, después de la ejecución exitosa del comando `decKey.sh`.

Con la finalidad de simular cuando los archivos resumen son diferentes, hemos copiado el archivo `PK12.hash` a `PK12nok.hash` y le modificamos el primer caracter. El resultado se muestra en la Figura 5.10.

5.3 Implementación de protocolos

Los dos protocolos principales objeto de esta tesis (autenticación y cifrado), los describimos a detalle en el Capítulo 4. Veremos ahora como los implementamos.

En general y con la idea de poder realizar pruebas y mostrar el correcto funcionamiento de estos protocolos, realizamos la implementación cliente-servidor en el mismo equipo, pero usando una comunicación TCP/IP por sockets. Así, si se requiriera probar estos protocolos en dos computadores en red, lo único que se tendría que cambiar, del lado del cliente, sería el nombre (o dirección IP) del equipo con el que se realizaría la comunicación, ya que en los programas que describiremos a continuación, la dirección que se tiene puesta, es la dirección local del equipo donde se esta

Figura 5.10: Ejecución del programa `decKey.sh` con un resultado no exitoso.

corriendo el programa. El puerto que se configuró cuando el programa funciona como servidor, es el 12345 y en el caso del protocolo de autenticación como veremos más adelante, los dos equipos pueden funcionar como servidores y en ese caso, y sólo con la finalidad de tomar números diferentes, se usa el puerto 12346. Por otra parte y considerando que se busca tener una serie de programas funcionales, estos programas se corrieron considerando que previamente se habían generado las claves pública y privada correspondientes a polinomios con 32 variables que generan 10 polinomios.

Para apreciar mejor la correcta implementación de estos protocolos, es conveniente tener dos terminales abiertas al mismo tiempo. En una se levantará el programa

Archivo	Descripción
`PK32.ir`	Archivo con una clave pública en su representación en enteros para 32 variables
`uov.sage`	Biblioteca de funciones para esquema AVNE
`verifier.sage`	Programa servidor que ejecuta el `verificador`
`verifier.sage.py`	Archivo generado automáticamente por Sage, con las adecuaciones de `verifier.sage`

Tabla 5.4: Contenido del directorio `AuthProt/Verifier`

servidor (que para el caso del protocolo de autenticación es el *verificador* y para el protocolo de cifrado corresponde a *Beto*) y en la otra, el programa cliente (aquí para el protocolo de autenticación será el *probador* y para el de cifrado *Alicia*). Levantaremos primero el servidor observando que se queda en espera de una conexión y después se levanta el cliente, observando la conexión que se establece entonces entre el servidor y cliente, iniciando así el protocolo correspondiente.

5.3.1 Implementación del protocolo de autenticación

Bajo el directorio `AuthProt`, del enlace `Programas`, se encuentran dos directorios `Verifier` y `Prover`. En estos directorios, se encuentran los archivos a usar como ejemplo de la implementación de este protocolo. Es importante recordar que las claves se generaron para diez polinomios en 32 variables.

Autenticación: el *verificador*

La Tabla 5.4 muestra los archivos que deben existir en `AuthProt/Verifier`, así como la función de estos.

El *verificador*, realizará la función de servidor, y sólo debe contar con la información de la llava pública, en este caso el archivo `PK32.ir`. Los archivos adicionales mostrados en la Tabla 5.4 y como ahí mismo se explica, son los necesarios para correr el programa como servidor.

El programa `verifier.sage` realiza las siguientes funciones:

1. Define al puerto 12345 para escuchar por él peticiones externas y el uso del puerto 12346 para comunicarse con un cliente que en este caso será el puerto que un *probador* deberá levantar para permitir conexiones desde el *verificador*.

2. Define todas las estructuras de datos para poder leer el archivo `PK32.ir` y tener internamente definidos los polinomios que forman la clave pública.

3. Genera aleatoriamente la imagen que desea cumplan los polinomios de la clave pública y se queda en espera a que se conecte el cliente que desea autenticarse, es decir el *probador*.

4. Cuando se conecta el *probador* le manda la imagen generada antes y cierra la conexión.

5. Aleatoriamente genera un número que representará el número de polinomios de la clave pública que combinará linealmente, generando un nuevo polinomio que enviará al *probador* en cuanto este se vuelva a conectar. Espera esta conexión por parte del *probador* y cuando ésta se establece, envía al *probador*, el polinomio del que deberá obtener la imagen que produce.

6. Como el *verificador* generó la imagen deseada en 3 y sabe cuales polinomios generaron el polinomio enviado al *probador*, sabe cuales elementos del vector imagen están involucrados y por lo tanto el valor al que debe evaluar el polinomio generado. Este valor lo manda a pantalla sólo como referencia pero nunca es enviado al *probador*.

7. El *verificador* inicia una conexión ahora como cliente del *probador* esperando que le mande el valor de la imagen del polinomio que generó y envió al *probador*.

8. Cuando recibe este valor, lo compara contra el esperado y si son diferentes lo registra en una bandera.

9. Repite los pasos 3 a 8 diez veces (esto sólo como ejemplo).

10. Revisa el valor de la bandera y si tiene registrado que hubo error (no importa cuantas veces) entonces rechaza la conexión, de otra forma acepta.

Autenticación: el *probador*

Por el lado del *probador*, los archivos que requiere para su correcta operación se encuentran en el directorio `AuthProt/Prover` y en la Tabla 5.5 se relacionan éstos, junto con la función de cada uno. Es importante notar, que el *probador* poseerá los archivos de la clave privada.

Las funciones que realiza el programa `prover.sage` son las siguientes:

1. Define el puerto 12346 para escuchar por él peticiones externas y el uso del puerto 12345 para comunicarse con un servidor, en este caso con el *verificador*.

2. Define las estructuras de datos necesarias para poder manejar la clave privada, formada por los polinomios, matriz y vector de la transformación afín. Una vez realizadas estas definiciones, lee el archivo con la representación entera de los polinomios de la clave privada (`SK32.ir`) y después el archivo con la representación entera de la matriz y vector de la transformación afín (`SK32.mv`).

3. Realiza ahora una conexión con el *verificador* a quien espera encontrar en el puerto 12345. De las actividades que realiza el *verificador*, vemos que este le manda la imagen directa que desea cumplan los polinomios que forman la clave pública. El *probador* utilizando los elementos que tiene como clave secreta,

Archivo	Descripción
`PK32.ir`	Archivo con una clave pública en su representación en enteros para 32 variables
`proverNOK.sage`	Programa cliente que ejecuta el *probador* para simular que no genera información correcta
`proverNOK.sage.py`	Archivo generado automáticamente por Sage, con las adecuaciones de `proverNOK.sage`
`prover.sage`	Programa cliente que ejecuta el *probador* en operación normal
`prover.sage.py`	Archivo generado automáticamente por Sage, con las adecuaciones de `prover.sage`
`SK32.ir`	Archivo con la representación en enteros de los polinomios en 32 variables que forman la clave secreta
`SK32.mv`	Archivo con la representación en enteros de la matriz y vector que forman la transformación afín, parte de la clave secreta
`uov.sage`	Biblioteca de funciones para esquema AVNE

Tabla 5.5: Contenido del directorio `AuthProt/Prover`

encuentra la imagen inversa de la información recibida, en este caso bajo los polinomios que forman la clave privada y luego, usando la transformación afín, encuentra los valores de la imagen inversa bajo los polinomios públicos.

4. El *probador* realiza nuevamente otra conexión con el *verificador* ahora para recibir el polinomio generado como una combinación lineal de algunos de los polinomios de la clave pública. Después de recibirlo, sustituye en él, los valores de la imagen inversa calculados en el punto 3 y ahora queda en espera que el *verificador* se conecte para enviarle el resultado calculado.

5. El *verificador* se conecta con el *probador* buscándolo en el puerto 12346 para que le de el valor calculado al que evalúa el polinomio enviado.

6. En este ejemplo, repite los puntos 3 a 5 diez veces y el programa termina. Es el *verificador* quien decide si acepta o no al *probador*.

Autenticación: ejecución y resultados

Como mencionamos al inicio de la Sección 5.3, para apreciar la ejecución de este protocolo, es necesario abrir dos terminales simultáneamente. En una correremos el programa para el *verificador* y en la otra el del *probador*. Con esta consideración presente, el programa que se debe correr primero, es el del *verificador* y una vez que éste quede en estado de espera se correrá el programa del *probador*, para que se inicie la conexión correspondiente y se vea correr de forma completa el protocolo.

Para ejecutar el programa servidor, del *verificador*, cambiarse al directorio `AuthProt/Verifier` y ejecutar ahí el siguiente comando:

```
sage verifier.sage
```

Con esto, se levanta el programa servidor del *verificador* y después de realizar los pasos que se mencionan en la Sección 5.3.1, se queda en espera de recibir alguna conexión desde un *probador* que desea autenticarse. La Figura 5.11 muestra la salida del programa quedando en espera.

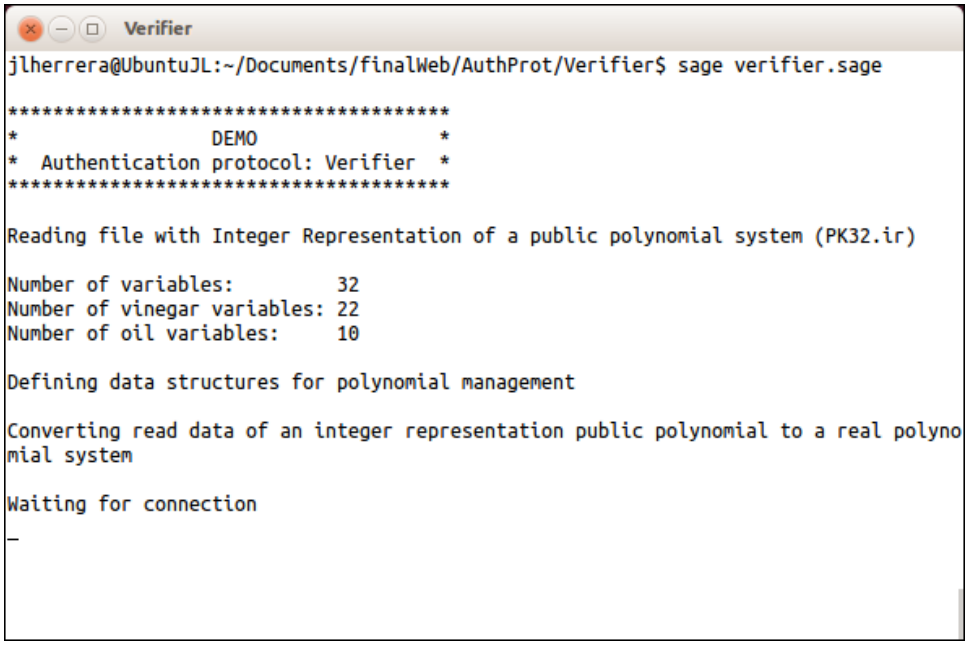

Figura 5.11: Ejecución del programa servidor `verifier.sage` ejecutado por el *verificador*, que queda en espera de que un *probador* se conecte.

Ahora hay que levantar el programa del lado del *probador*. Para esto, usar otra terminal y cambiarse al directorio `AuthProt/Prover`; ejecutar en ella el comando:

```
sage prover.sage
```

El programa del lado del *probador* iniciará y se empezarán a realizar los pasos mencionados en la Sección 5.3.1. La Figura 5.12 muestra los mensajes que se obtienen al correr inicialmente este programa.

A partir de este momento, los dos programas empiezan a interactuar como se ha descrito antes. En la Figura 5.13 podemos apreciar la actividad en el lado del *verificador* para los primeros dos ciclos de interacción con el *probador* y de manera similar, las acciones ejecutadas por el probador para los mismos dos ciclos, se aprecian en la Figura 5.14.

Entrando al detalle de esta interacción, vemos lo siguiente:

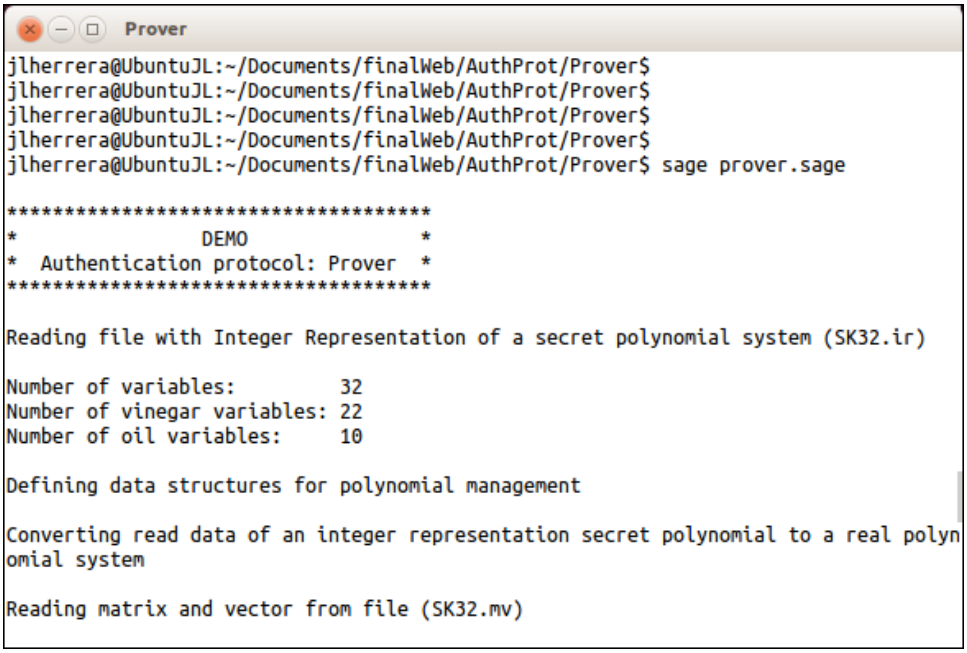

Figura 5.12: Ejecución del programa cliente `prover.sage` ejecutado por el *probador*, que inicia interacción con el *verificador*.

1. El *verificador* manda el mensaje `1. Got connection from ('127.0.0.1', 41189). Sending 'y': (0, 0, 1, 0, 1, 1, 0, 1, 1, 0)`, que indica desde que IP y puerto recibe una conexión e inmediatamente despliega la imagen $\mathbf{y}$ que desea cumplan los polinomios públicos $\mathcal{P}$. Muestra después el mensaje: `Retries to find combined polynomial: 2` que reporta el número de intentos que realizó el programa, para encontrar p_{LC}, producto de la combinación lineal de algunos polinomios de la llave pública, donde el término principal del polinomio generado, no existe en ninguno de los términos principales de los polinomios públicos (ver Sección 4.1.3 para más detalle). Queda en espera de la respuesta del *probador*.

2. El *probador* se conecta con el *verificador* y muestra el valor de $\mathbf{y}$ que recibe: `1. Received y: (0, 0, 1, 0, 1, 1, 0, 1, 1, 0)`. El *probador*, realiza el cómputo de $\mathbf{x}$ y despliega: `Iterations made to find solution: 6`, que corresponde al número de iteraciones que realizó para encontrar la imagen inversa $\mathbf{x}$ de $\mathbf{y}$ bajo los polinomios públicos $\mathcal{P}$. El *probador* vuelve a realizar otra conexión con el *verificador*.

3. Al recibir otra conexión, el *verificador*, envía a la terminal: `1. Got connection from ('127.0.0.1', 41190). Sending p_LC: [113830113, 412466297, 969639495, 280059239, 215794012, 123069866, 60100359, 18954085, 12110089, 8146980, 3939719, 1558676, 974451, 239427, 74181, 79323, 49011, 696, 2038, 3107, 3302, 1289, 286, 455, 111, 78, 57, 26, 4, 4, 0, 1, 3000676373, 0]`,

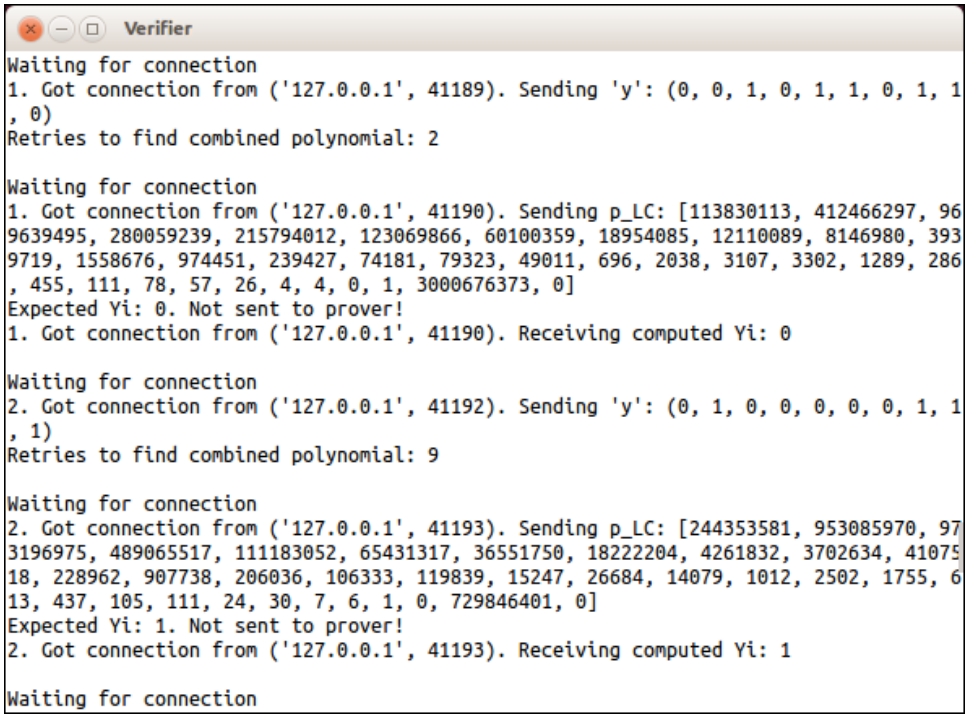

Figura 5.13: Interacción *verificador-probador* del lado del *verificador* para los primeros dos ciclos.

que corresponde a la representación entera del polinomio, producto de la combinación lineal de polinomios de la clave pública, seleccionados al azar, para que el *probador* regrese el resultado de la imagen de dicho polinomio cuando se evalúa en los valores encontrados de **x**. Se muestra también como salida en la terminal del *verificador*, el valor que espera le regrese el *probador*: `Expected Yi: 0. Not sent to prover!`, pero este valor NO se envía, sólo es como referencia en este programa de demostración, para saber que debe responder el *probador*.

4. El *probador*, recibe el polinomio generado por el *verificador* y esto lo vemos cuando el *probador* muestra en la terminal: `1. Received p_LC: [113830113, 412466297, 969639495, 280059239, 215794012, 123069866, 60100359, 18954085, 12110089, 8146980, 3939719, 1558676, 974451, 239427, 74181, 79323, 49011, 696, 2038, 3107, 3302, 1289, 286, 455, 111, 78, 57, 26, 4, 4, 0, 1, 3000676373, 0]`. Entonces el *probador* encuentra el valor de `Yi` mostrando el mensaje: `Computed bit: 0`, y queda en espera (mensaje: `Waiting for connection`) que se conecte el *verificador*, para enviarle este valor.

5. El *verificador* se conecta al *probador*[15] para obtener el valor de `Yi` calculado por este último: `1. Got connection from ('127.0.0.1', 41190). Receiving computed`

[15]Es en este caso cuando el *probador* opera como servidor y el *verificador* como cliente

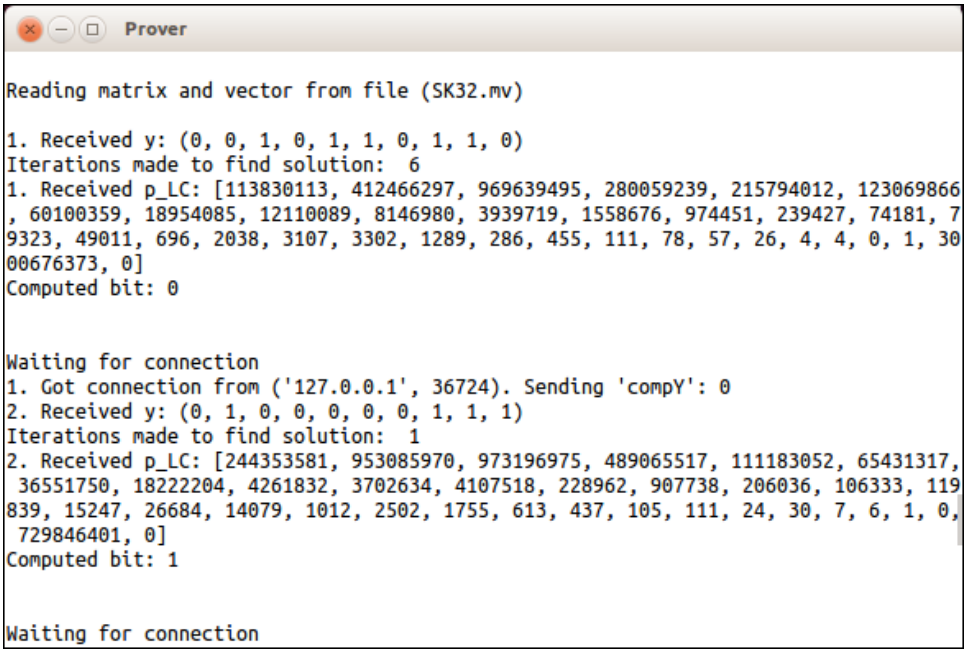

Figura 5.14: Interacción *probador-verificador* del lado del *probador* para los primeros dos ciclos.

`Yi: 0`.

6. Del lado del *probador*, vemos esta conexión, cuando despliega: `1. Got connection from ('127.0.0.1', 36724). Sending 'compY': 0`.

7. Aquí termina el primer ciclo y en esta demostración, se realizarán nueve ciclo más, y si todos ellos son exitosos, el *verificador* mandará un mensaje de aceptación al final.

En la Figura 5.15, se muestra el último ciclo que ejecuta el *verificador* y en este caso, los diez ciclos han sido exitosos, por lo que en la terminal del *verificador* se muestra el mensaje `Authentication passed? True`, que es el resultado de verificar que todos los retos fueron respondidos satisfactoriamente. El ciclo de autenticación termina aquí y puede iniciarse uno nuevo si así se desea.

Con la finalidad de revisar, que el protocolo puede detectar casos en que el *probador* sólo esté **adivinando** los valores de la imagen `Yi` del polinomio que el *verificador* generó y envió al *probador*, se realizó una modificación al programa `prover.sage`. Esta modificación se grabó en el archivo `proverNOK.sage` que también se encuentra en el directorio `AuthProt/Prover` del enlace Programas. Cuando se corre este programa[16], se obtiene una salida como la de la Figura 5.16, en donde se muestra el término del programa `verifier.sage`, mostrando que no se acepta la autenticación, esto debido

[16]Como en el caso del programa sin modificación, primero es necesario correr el programa del *verificador* en una ventana (`sage verifier.sage` y después en otra correr el programa del *probador* modificado: `sage proverNOK.sage`

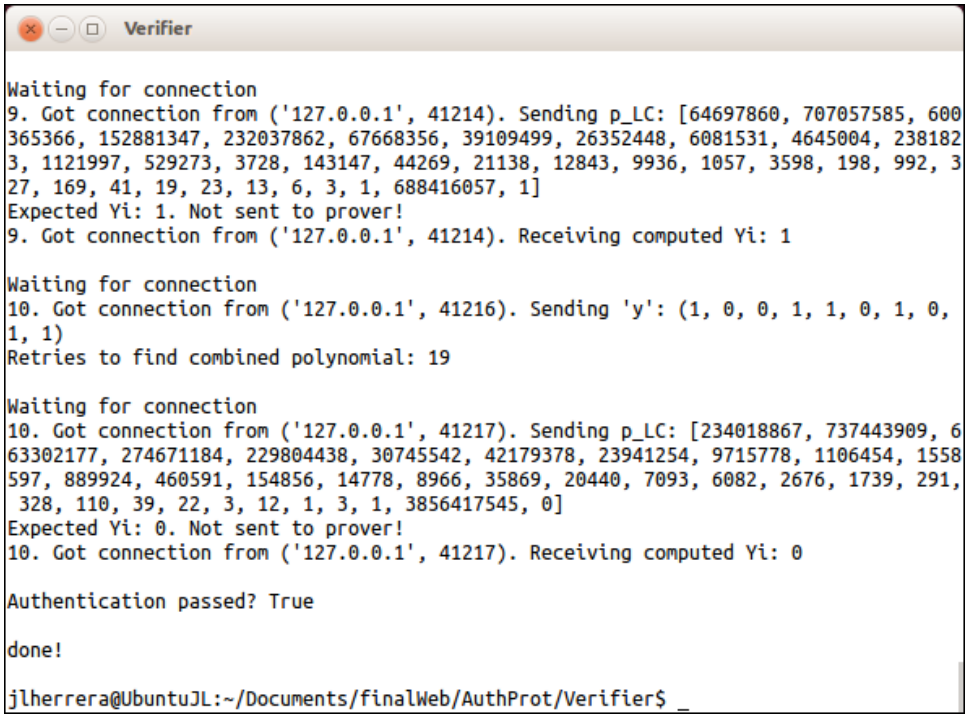

Figura 5.15: Último ciclo ejecutado por el *verificador*, en este caso aceptando al *probador*.

a que hubo valores de `Yi` erróneos, por ejemplo, el último valor que se espera es 1[17], pero el valor de `Yi` que se muestra se recibió es 0[18]. De hecho, hay más errores antes, pero en la Figura 5.16 sólo se muestra la parte final de la salida. Al terminar el programa, sólo se dice si se aceptó o no la identidad del *probador*, pero no se dan detalles de en donde falló, con la finalidad de no dar información adicional a alguien más que pudiera estar observando la salida del *verificador*.

La modificación que se realizó al programa `prover.sage`, para dejarla en el programa `proverNOK.sage`, se muestra en la Figura 5.17, y es el resultado de la ejecución del comando `diff`. Lo que se realizó fue comentar la línea que realiza el cómputo de la imagen del polinomio que generó el *verificador* y que recibió el *probador* en `rxPoly`, por la generación aleatoria de un número 0 o 1: se cambió `compY = evalPoly(rxPoly, x, xi)` por `compY = ZZ.random_element(2)`. Con esto se simula que el *probador* este sólo adivinando los valores de `Yi` y para este caso de diez ciclos, vemos que no pasa la prueba[19], ya que el mensaje final del progama `verifier.sage` así lo muestra: `Authentication passed? False`.

Las pruebas anteriores, nos permiten observar como el protocolo de autenticación funciona adecuadamente y hacemos hincapié en el hecho de que el *probador* nunca manda los valores calculados de la imagen inversa del polinomio $\mathcal{P}$, es decir los valores

[17] El programa muestra en la Figura 5.16: `Expected Yi: 1. Not sent to prover!`

[18] Ver el mensaje de la Figura 5.16: `10. Got connection from ('127.0.0.1', 41247). Receiving computed Yi: 0`

[19] La probabilidad de pasar las diez pruebas en este caso es de $1/2^{10} = 0.0009765$.

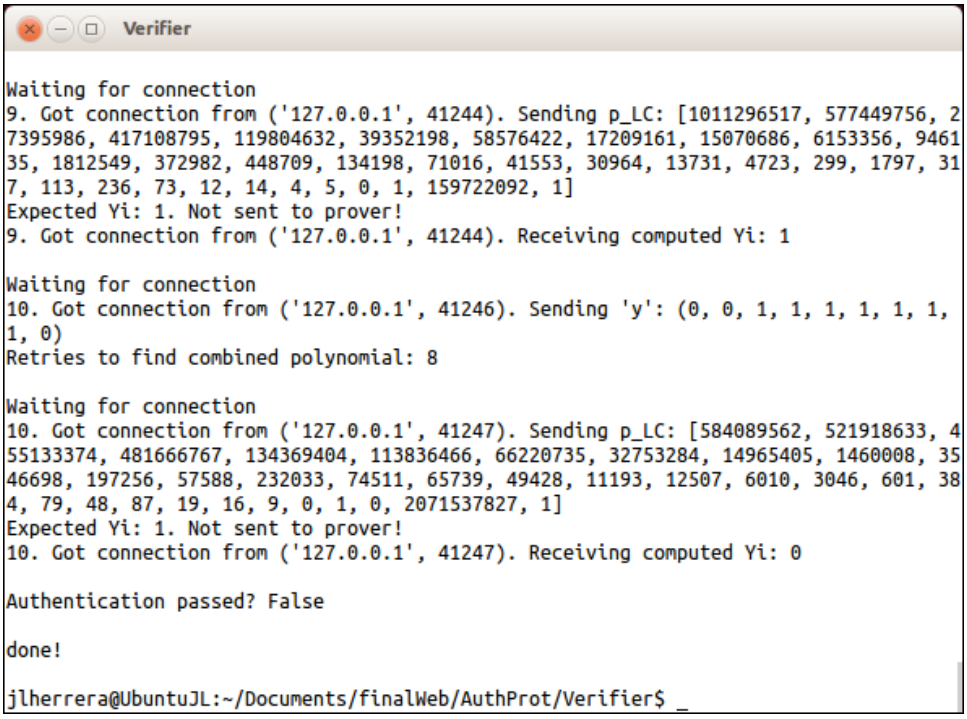

Figura 5.16: Último ciclo ejecutado por el *verificador*, cuando el *probador* sólo **adivina** los valores de `Yi`.

de **x**. El *probador* sólo manda al *verificador* el valor de la imagen del polinomio que es generado por el mismo *verificador* y quien de antemano sabe el valor correcto que debe enviarle el *probador*. Dado que la interacción *verificador-probador* puede entonces simularse completamente por un algoritmo, afirmamos que este protocolo de autenticación es de conocimiento nulo perfecto (ver Definición 2.4.1).

5.3.2 Implementación del protocolo de cifrado

Bajo el directorio `EncProt`, del enlace `Programas`, se encuentran dos directorios `Alicia` y `Beto`. En estos directorios, se encuentran los archivos a usar como ejemplo de la implementación de este protocolo. Como en el caso del protocolo de autenticación, es importante recordar que las claves a usar en esta demostración, se generaron para diez polinomios en 32 variables.

Cifrado: el servidor *Beto*

En la Tabla 5.6 se muestran los archivos que deben existir en `AuthProt/Verifier`, así como la función de estos.

En este caso, *Beto* realizará la función de servidor y sólo debe contar con la información de la clave pública, en este caso el archivo `PK32.ir`. Los archivos adicionales mostrados en la Tabla 5.6 y como ahí mismo se explica, son los necesarios para correr

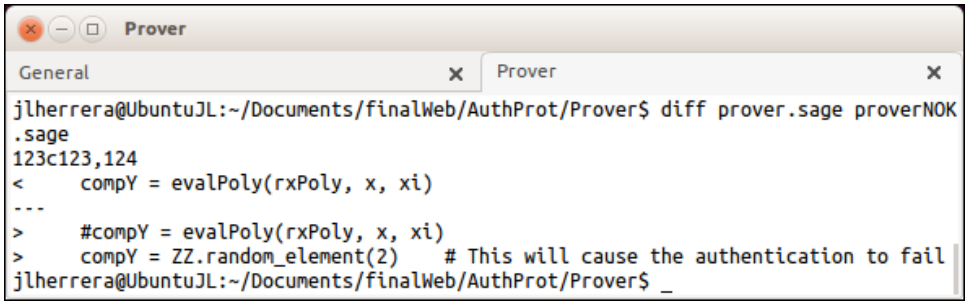

Figura 5.17: Diferencias entre el programa `prover.sage` y `proverNOK.sage`.

Archivo	Descripción
`PK32.ir`	Archivo con una clave pública en su representación en enteros para 32 variables
`server.sage`	Programa servidor que ejecuta `Beto`
`server.sage.py`	Archivo generado automáticamente por Sage, con las adecuaciones de `server.sage`
`uov.sage`	Biblioteca de funciones para esquema AVNE

Tabla 5.6: Contenido del directorio `EncProt/Beto`

el programa como servidor.

El programa `server.sage` realiza las siguientes funciones:

1. Define al puerto 12345 para escuchar por él peticiones externas.

2. Lee la clave pública en su representación en enteros, obteniendo además el total de variables n y el número de polinomios m que es igual al número de variables aceite.

3. Genera aleatoriamente una cadena de m bits, en este caso $m = 10$, que se usará como la cadena de bits a cifrar.

4. Define todas las estructuras de datos para poder convertir el archivo `PK32.ir` a los polinomios en su representación natural, que forman la clave pública.

5. Inicializa un apuntador al primer elemento de la cadena a cifrar

6. Genera aleatoriamente la imagen que desea cumplan los polinomios de la clave pública y se queda en espera a que se conecte *Alicia* para enviarle la imagen generada.

7. Cuando se conecta *Alicia*, le manda la imagen generada antes y cierra la conexión.

8. Obtiene el valor del bit donde señala el apuntador a la cadena de caracteres.

Archivo	Descripción
`client.sage`	Programa cliente que ejecuta *Alicia* en operación normal
`client.sage.py`	Archivo generado automáticamente por Sage, con las adecuaciones de `client.sage`
`PK32.ir`	Archivo con una clave pública en su representación en enteros para 32 variables
`SK32.ir`	Archivo con la representación en enteros de los polinomios en 32 variables que forman la clave secreta
`SK32.mv`	Archivo con la representación en enteros de la matriz y vector que forman la transformación afín, parte de la clave secreta
`uov.sage`	Biblioteca de funciones para esquema AVNE

Tabla 5.7: Contenido del directorio `EncProt/Alicia`

9. Aleatoriamente, genera un número que representará el número de polinomios de la clave pública a combinar linealmente. Realiza esta combinación lineal, asegurándose además que este nuevo polinomio tenga como imagen el valor del bit obtenido en el punto 8. Verifica además que el término principal del polinomio generado, no sea igual a ninguno de los términos principales de los polinomios que forman la clave pública (ver más detalles en la Sección 4.2.2). Este nuevo polinomio lo enviará a *Alicia* en cuanto se vuelva a conectar. Espera esta conexión por parte de *Alicia* y le envía dicho polinomio en su representación en enteros, con lo que manda el bit correspondiente de la cadena de caracteres a cifrar, cifrado como un polinomio.

10. Incrementa el apuntador de la cadena de caracteres a cifrar.

11. Repite los pasos 6 a 10 hasta procesar todos los bits de la cadena a cifrar.

Descifrado: el cliente *Alicia*

Por el lado del cliente, *Alicia*, los archivos que requiere para su correcta operación se encuentran en el directorio `EncProt/Alicia` y en la Tabla 5.7 se relacionan éstos, junto con la función de cada uno. Es importante notar, que el cliente *Alicia* poseerá los archivos de la clave privada.

Las funciones que realiza el programa `client.sage` son las siguientes:

1. Define las estructuras de datos necesarias para poder manejar la clave privada, formada por los polinomios, matriz y vector de la transformación afín. Una vez realizadas estas definiciones, lee el archivo con la representación entera de los polinomios de la clave privada (`SK32.ir`) y después el archivo con la representación entera de la matriz y vector de la transformación afín (`SK32.mv`).

2. Realiza ahora una conexión con el servidor *Beto* a quien espera encontrar en el puerto 12345. De las actividades que realiza el servidor *Beto*, vemos que este le manda la imagen directa que desea cumplan los polinomios que forman la clave pública. El cliente *Alicia*, utilizando los elementos que tiene como clave secreta, encuentra la imagen inversa de la información recibida, en este caso bajo los polinomios que forman la clave privada y luego, usando la transformación afín, encuentra los valores de la imagen inversa bajo los polinomios públicos.

3. *Alicia* realiza nuevamente otra conexión con el servidor *Beto*, ahora para recibir el polinomio generado como una combinación lineal de algunos de los polinomios de la clave pública. Después de recibirlo, sustituye en él, los valores de la imagen inversa calculados en el punto 2 y el valor al que evalúa esta sustitución, corresponderá al bit descifrado.

4. En este ejemplo, repite los puntos 2 a 3 diez veces y el programa termina. Con esto, el cliente *Alicia* ha descifrado completamente la cadena de $m = 10$ bits que se cifraron por medio de polinomios.

Cifrado: ejecución y resultados

Como mencionamos al inicio de la Sección 5.3, para apreciar la ejecución de este protocolo, es necesario abrir dos terminales simultáneamente. En una correremos el programa para el servidor *Beto* y en la otra el cliente *Alicia*. Con esta consideración presente, el programa que se debe correr primero, es el del servidor *Beto* y una vez que éste quede en estado de espera se correrá el programa del cliente *Alicia*, para que se inicie la conexión correspondiente y se vea correr de forma completa el protocolo.

Para ejecutar el programa del servidor *Beto* cambiarse al directorio `EncProt/Beto` y ejecutar ahí el siguiente comando:

```
sage server.sage
```

Con esto, se levanta el programa servidor de *Beto* y después de realizar los pasos que se mencionan en la Sección 5.3.2, se queda en espera de recibir alguna conexión desde un cliente al que se enviará el cifrado de una cadena de caracteres. La Figura 5.18 muestra la salida del programa quedando en espera. Observar que en esta figura, se muestra también la cadena a cifrar, en este caso con el texto: `String to encrypt: (1, 1, 0, 0, 0, 0, 0, 0, 1, 0)`, misma que con fines de demostración, se generó aleatoriamente.

Ahora hay que levantar el programa del lado del cliente *Alicia*. Para esto, usar otra terminal y cambiarse al directorio `EncProt/Alicia`. Una vez ahí, ejecutar el comando:

```
sage client.sage
```

El programa del lado del cliente *Alicia* iniciará y se empezarán a realizar los pasos mencionados en la Sección 5.3.2. La Figura 5.19 muestra los mensajes que se obtienen al correr inicialmente este programa.

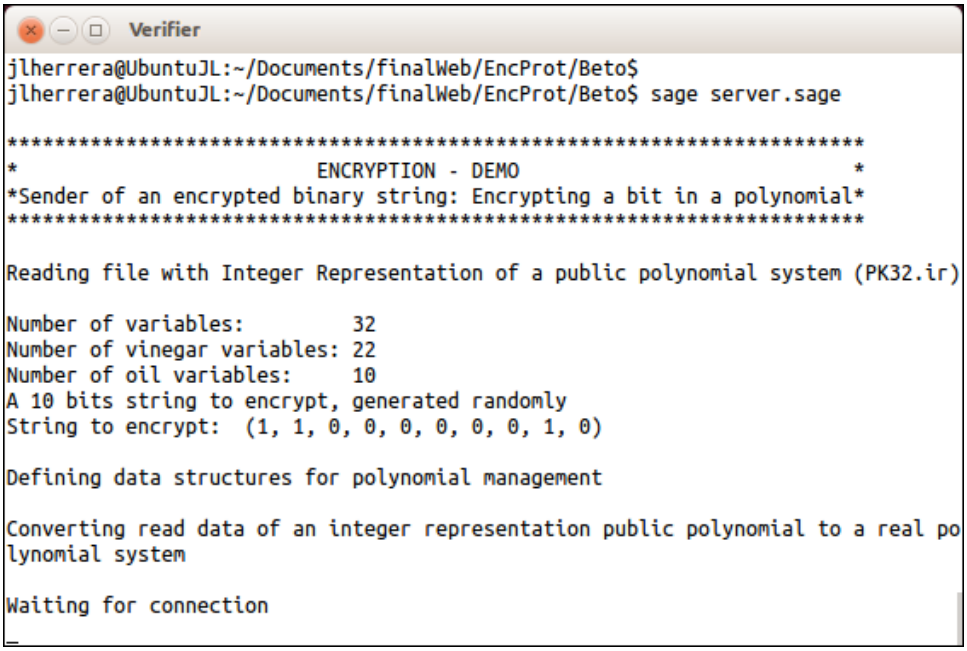

Figura 5.18: Ejecución del programa servidor `server.sage` ejecutado por *Beto*, el cual queda en espera de que un cliente *Alicia* se conecte.

A partir de este momento, los dos programas empiezan a interactuar como se ha descrito antes. En la Figura 5.20 podemos apreciar la actividad en el lado del servidor *Beto* interactuando con el cliente *Alicia*, para los primeros dos bits de la cadena a cifrar. De manera similar, las acciones ejecutadas por el cliente *Alicia* para los mismos dos bits, interactuando con el servidor *Beto*, se aprecian en la Figura 5.21.

Entrando al detalle de esta interacción, vemos lo siguiente:

1. El servidor *Beto* muestra el mensaje `1. Got connection from ('127.0.0.1', 41289). Sending 'y': (0, 1, 0, 1, 0, 1, 0, 1, 1, 1)`, que indica desde que IP y puerto recibe una conexión e inmediatamente despliega la imagen $\mathbf{y}$ que desea cumplan los polinomios públicos $\mathcal{P}$. Muestra además cuantas iteraciones realizó para encontrar el polinomio p_{LC} que enviará posteriormente a *Alicia*: `Retries to find combined polynomial: 40`. Después, queda en espera de la respuesta del cliente *Alicia*.

2. El cliente *Alicia* se conecta con el servidor *Beto* y muestra el valor de $\mathbf{y}$ que recibe: `1. Received y: (0, 1, 0, 1, 0, 1, 0, 1, 1, 1)`. El cliente *Alicia*, realiza el cómputo de $\mathbf{x}$ y despliega ahora `Iterations made to find solution: 1`, que corresponde al número de iteraciones que realizó para encontrar la imagen inversa $\mathbf{x}$ de $\mathbf{y}$ bajo los polinomios públicos $\mathcal{P}$. El cliente *Alicia* vuelve a realizar otra conexión con el servidor *Beto*.

3. Al recibir otra conexión, el servidor *Beto*, envía `1. Got connection from ('127.0.0.1', 41290). Sending p_LC: [175356565, 146976690, 328152282, 293835416,`

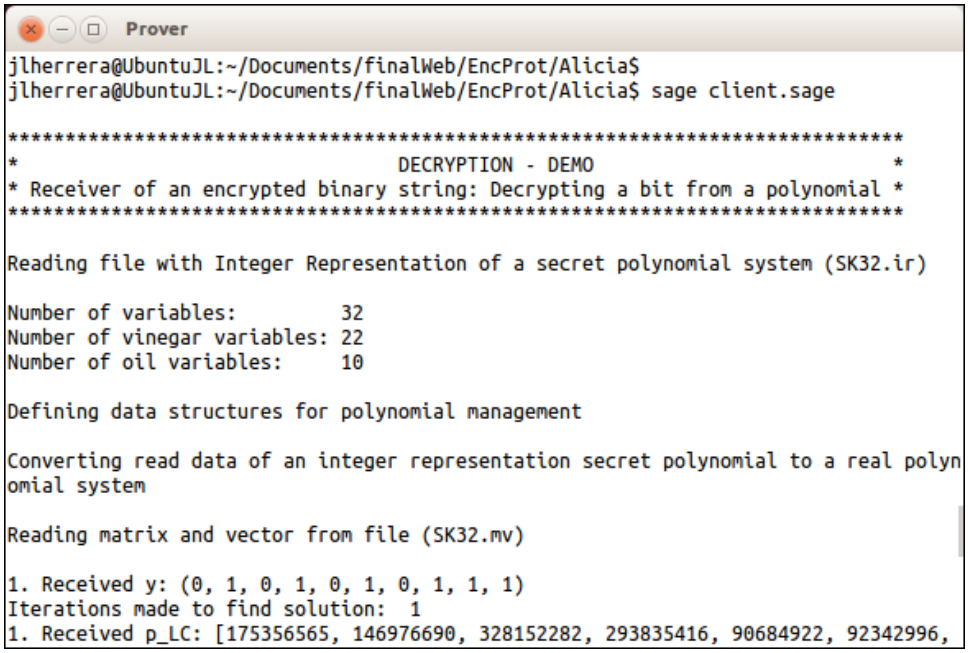

Figura 5.19: Ejecución del programa `client.sage` ejecutado por *Alicia*, que inicia interacción con *Beto*.

`90684922, 92342996, 44148000, 31588111, 14047845, 5714515, 3781624, 1558487, 1003196, 140802, 98572, 18864, 63591, 28654, 14167, 4350, 3795, 1193, 984, 249, 68, 17, 48, 8, 1, 0, 2, 1, 1139358609, 0]`, que corresponde a la representación entera del polinomio, producto de la combinación lineal de polinomios de la clave pública, seleccionados al azar, pero que evalúa al valor del bit que se envía cifrado como un polinomio.

4. El cliente *Alicia*, recibe el polinomio generado por el servidor *Beto*. Esto lo vemos cuando el cliente *Alicia* muestra en la terminal: `1. Received p_LC: [175356565, 146976690, 328152282, 293835416, 90684922, 92342996, 44148000, 31588111, 14047845, 5714515, 3781624, 1558487, 1003196, 140802, 98572, 18864, 63591, 28654, 14167, 4350, 3795, 1193, 984, 249, 68, 17, 48, 8, 1, 0, 2, 1, 1139358609, 0]`. Entonces el cliente *Alicia* encuentra el valor al que evalúa el polinomio recibido cuando se sustituyen en él, los valores de la imagen inversa calculada antes, mostrando el mensaje: `Decrypted bit: 1`.

5. Este proceso continua hasta que el servidor *Beto*, haya procesado todos los bits a cifrar y en ese momento, el cliente *Alicia*, muestra la cadena completa descifrada: `Decrypted string: (1, 1, 0, 0, 0, 0, 0, 0, 1, 0)`.

En la Figura 5.22, se muestra el descifrado de los bits 9 y 10 del total de diez a descifrar. Se muestra además la cadena completa descifrada, en la línea `Decrypted string: (1, 1, 0, 0, 0, 0, 0, 0, 1, 0)`, lo cual corresponde con el mensaje que

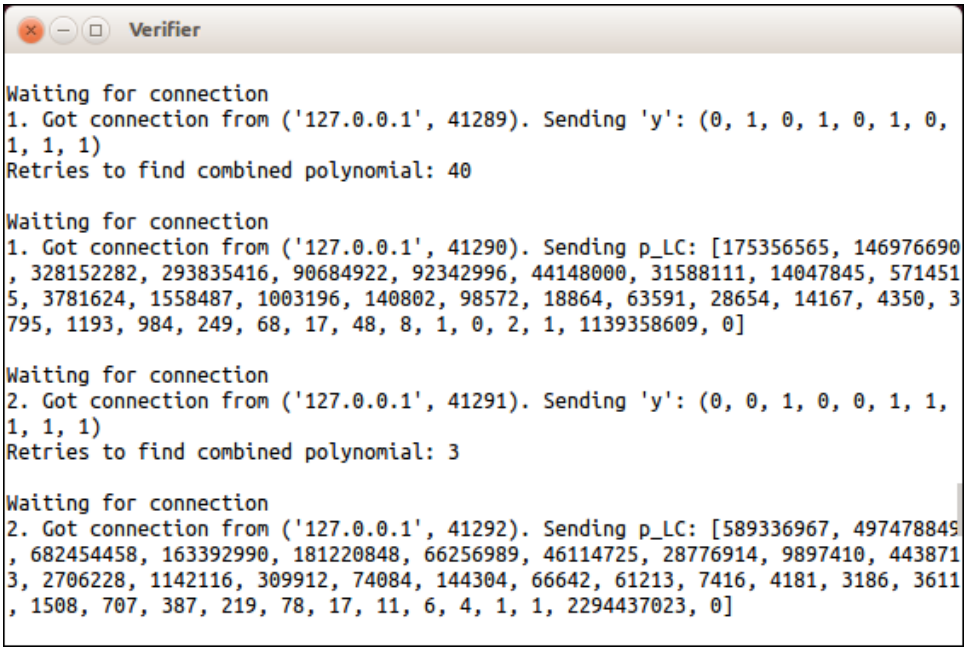

Figura 5.20: Interacción *Beto-Alicia* del lado del servidor *Beto* para los primeros dos bits de la cadena a cifrar.

se generó en el lado del servidor *Beto* (ver Figura 5.18). El ciclo de autenticación termina aquí y puede iniciarse uno nuevo si así se desea.

Las pruebas anteriores, nos permiten observar como el protocolo de cifrado funciona adecuadamente. Es importante resaltar, que el cifrado es bit a bit y cada bit se cifra al representarlo por un polinomio, que al evaluarse en los valores previamente calculados de $\mathbf{x}$, permiten encontrar el valor de dicho bit.

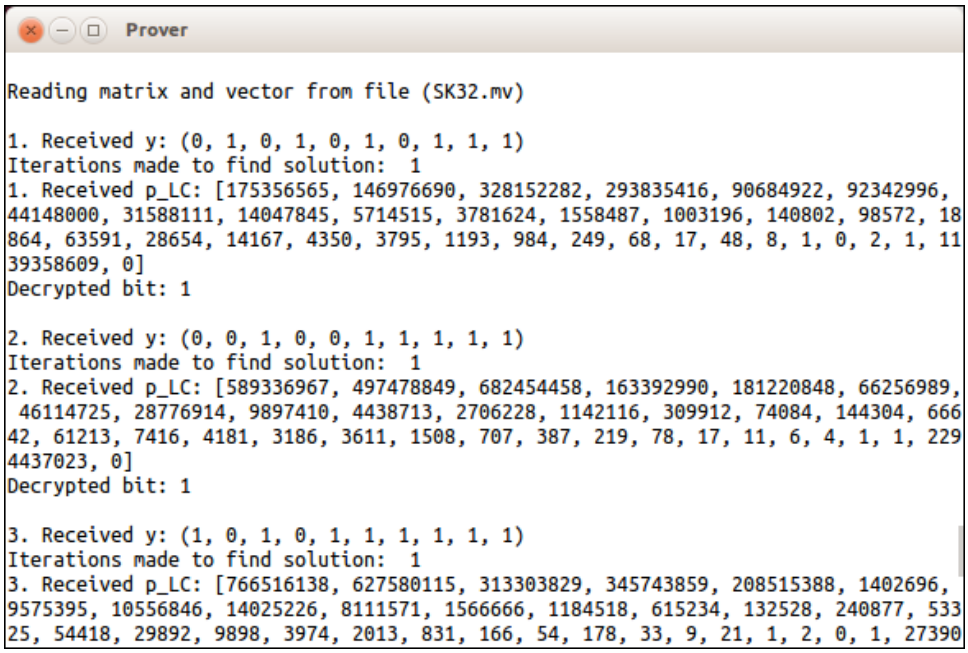

Figura 5.21: Interacción *Alicia-Beto* del lado del cliente *Alicia* para los primeros dos bits de la cadena cifrada.

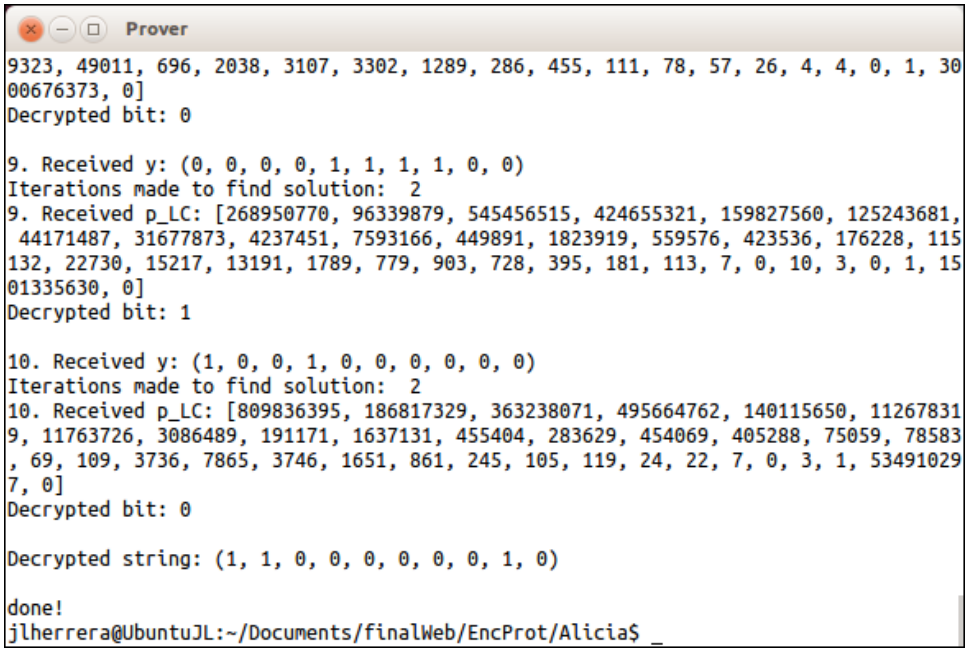

Figura 5.22: Parte final en el lado del cliente *Alicia* donde se puede observar la cadena completa descifrada.

Capítulo 6

Conclusiones y trabajo futuro

6.1 Conclusiones

Los planteamientos que originalmente realizamos tanto para el protocolo de autenticación como para el de cifrado se cumplen totalmente con las implementaciones que hemos realizado. En efecto, el protocolo de autenticación, muestra que el *verificador* puede asegurarse que un *probador* es quien dice ser:

- Encontrar la imagen inversa $\mathbf{x}$, del conjunto de Polinomios $\mathcal{P}$, para una imagen $\mathbf{y}$ generada aleatoriamente, es un problema difícil, cuando no se tiene una manera eficiente de calcular la función inversa.
- El *probador*, dado que conoce la clave secreta formada por los polinomios $\mathcal{P}'$, la matriz M_s y el vector $\mathbf{v_s}$ de la transformación afín, puede calcular la función inversa antes mencionada de manera eficiente.
- Cuando un *probador* no auténtico, pretende autenticarse por tan sólo adivinar los valores a los que evalúa el polinomio generado por el *verificador*, vemos que fácilmente falla. En los experimentos que realizamos, vimos que con tan sólo 10 ciclos de pruebas, no fue posible que dicho *probador* se autenticara exitósamente.

Para el caso del protocolo de cifrado, vimos que sólo *Alicia* que cuenta con la clave secreta, puede descifrar bit a bit el mensaje cifrado por *Beto*:

- *Beto*, conociendo la clave pública, puede encontrar eficientemente otro polinomio que represente un bit de la cadena a cifrar.
- Ese polinomio, evaluado en la imagen inversa que cumplen los polinomios públicos $\mathcal{P}$, representa un bit cifrado que *Beto* desea enviar.
- Sólo *Alicia* que tiene la clave secreta, puede descifrar eficientemente el valor de dicho bit.

Es importante recalcar, que la seguridad de estos protocolos, se basa en los siguientes puntos:

1. El esquema AVNE permanece seguro mientras la cantidad de variables vinagre sea del orden del doble de las variables aceite. Si esto no se cumple, el esquema AVNE es vulnerable.

2. La dificultad de encontrar cuáles polinomios de la clave pública, se combinaron linealmente para generar el polinomio p_{LC} que se envía al *probador* o bien a *Alicia*. En este caso, para evitar que el Algoritmo de la División para varias variables pudiera ayudar a encontrar estos polinomios, se genera p_{LC} asegurando que su término principal no sea ninguno de los que tienen los polinomios de la clave pública y con esto desde la primera iteración que realizara dicho algoritmo, se generaría un residuo con lo que este algoritmo no determinaría que polinomios se usaron.

3. Considerando que el esquema AVNE es seguro, la seguridad radica ahora en la dificultad de saber que polinomios se usaron para generar el polinomio que se manda como reto en el protocolo de autenticación al *probador* o bien el polinomio que representa un bit cifrado en el caso del protocolo de cifrado. Vimos entonces que un índice de seguridad de 128 bits, requiere claves de 384 variables para el protocolo de autenticación y 387 variables para el protocolo de cifrado, el primero en 128 polinomios y el último en 129.

Es importante también recordar, que los polinomios que se generan tanto públicos como privados, así como la matriz y vector de la transformación afín, se representan por números enteros, con lo cual, el tamaño de las claves privada y pública se disminuyen en una relación de alrededor de 15 veces. En la Tabla 6.1, se presentan los detalles que obtuvimos para polinomios públicos y privados desde 8 hasta 128 variables, donde el tamaño de las claves públicas (PK) o privadas (SK) esta en bytes, tanto para su representación natural como la representación en enteros.

No. Vars.	Vars. Aceite	Tamaño PK	Tamaño PK.ir	PK/PK.ir	Tamaño SK	Tamaño SK.ir	SK/SK.ir
8	2	327	57	6	271	57	5
9	3	575	99	6	430	96	4
10	3	813	113	7	507	104	5
15	5	2,790	316	9	2,250	297	8
16	5	2,939	340	9	2,414	321	8
20	6	5,775	579	10	5,171	548	9
32	10	25,489	2,085	12	21,717	1,940	11
48	16	91,437	6,802	13	79,282	6,196	13
64	21	215,598	15,116	14	185,429	13,743	13
96	32	736,862	49,245	15	644,992	44,403	15
128	42	1,789,349	111,929	16	1,561,952	101,083	15

Tabla 6.1: Tamaño de los archivos con claves pública y privada en su representación natural y en enteros.

Las Figuras 6.1 y 6.2 y tomando los datos de la Tabla 6.1, muestran gráficamente la conveniencia de representar en enteros estas claves, ya que como a su vez se ve en la Figura 6.3 el tamaño de los archivos con la representación en enteros ya sea de las claves públicas o privadas, es alrededor de 15 veces menor.

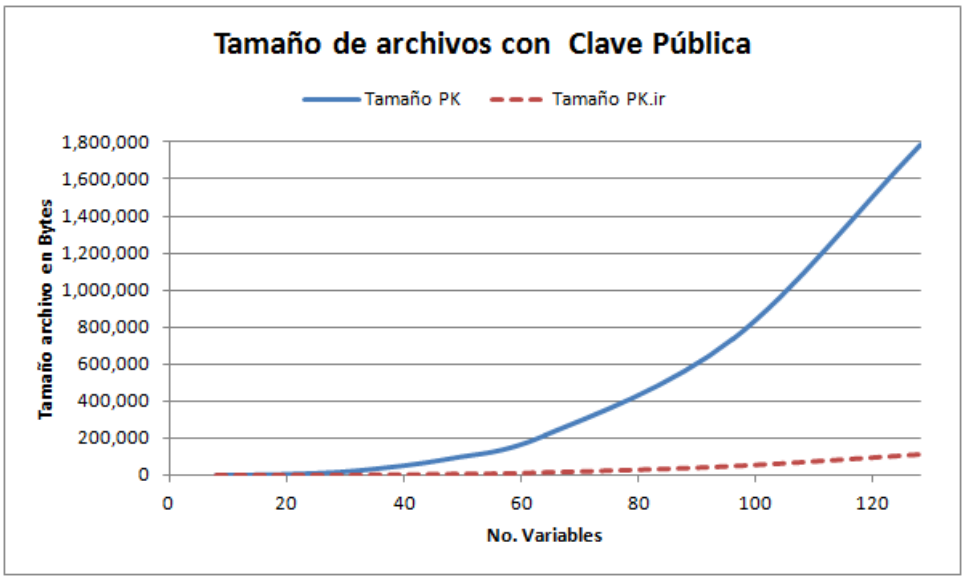

Figura 6.1: Tamaño de los archivos con la clave pública de forma natural y representada en enteros.

Revisando ahora el tema de los ataques algebráicos, es válido pensar en resolver el conjunto de polinomios que forman la clave pública por alguno de los métodos basados en las bases de Gröbner. En particular realizamos pruebas con PolyBoRi, un programa desarrollado para polinomios booleanos como el caso que nos ocupó en esta tesis y los resultados se muestran en la Tabla 6.2.

Ecs. Entrada	No. Variables	Ecs. Salida	Repeticiones	Tiempo Promedio
3	9	37	20	0.2509
3	10	47	20	0.5077
3	11	95	20	1.4999
4	12	103	20	2.9288
4	13	172	20	5.8067
4	14	280	20	29.1009
5	15	308	20	36.8047
5	16	475	20	306.1974
5	17	817	5	757.3434
6	18	830	1	2,147.1187
6	19	1,245	1	5,734.3288

Tabla 6.2: Tiempo para el cálculo de bases de Gröbner para polinomios generados por el método AVNE (Repeticiones es el número de veces que se corrió el programa con los mismos parámetros para calcular un tiempo promedio. El Tiempo Promedio esta en segundos).

Se usó un servidor Linux, con 4 cores disponibles y 64 GB de memoria. Como se puede apreciar, sólo se pudieron resolver hasta 6 polinomios con 19 variables, lo que

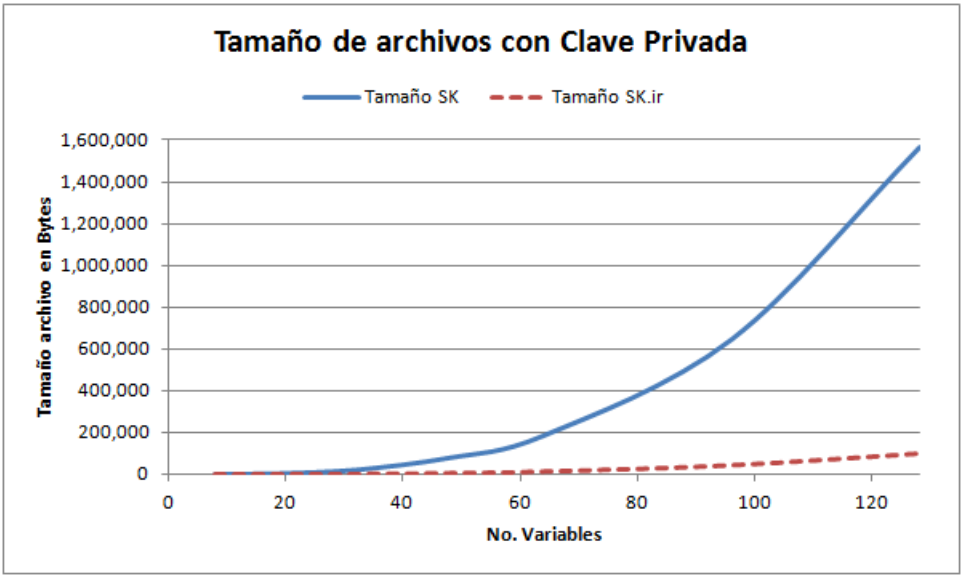

Figura 6.2: Tamaño de los archivos con la clave privada de forma natural y representada en enteros.

generó 1,245 ecuaciones con las que se pudieron obtener los valores de las variables que satisfacían esos seis polinomios en su variedad algebráica. El tiempo que tomó fue un poco más de hora y media, sin embargo, para 20 variables el problema que presentó fue que se agotó la memoria para que PolyBoRi pudiera elaborar sus estructuras internas de trabajo. En la Figura 6.4 vemos la gráfica del número de variables vs. el tiempo promedio obtenidos de la Tabla 6.2.

Con los datos de la Tabla 6.2 y observando el comportamiento de la Gráfica 6.4, encontramos una ecuación exponencial para proyectar el comportamiento del tiempo de solución por medio de bases de Gröbner de los polinomios que forman la clave pública de los protocolos de autenticación o cifrado. La ecuación es la siguiente:

$$t = 0.000121474 \exp^{0.996187x - 1.25646} \tag{6.1}$$

donde x representa al número de variables y t el tiempo de solución en minutos. La Figura 6.5 muestra la gráfica de la Ecuación 6.1 y tan sólo para polinomios en 34 variables, el tiempo proyectado de solución tomaría cientos de años, sin embargo el problema es también el uso exponencial de memoria ya que como dijimos antes, para polinomios en 20 variables no fue posible encontrar su base de Gröbner, por no haber memoria suficiente para las estructuras de trabajo de PolyBoRi.

Los protocolos funcionan como esperábamos, la seguridad que ofrecen se compara a la de RSA o ECC y en el presente documento se dan las explicaciones y resultados de los diferentes aspectos que consideramos importantes, sin embargo, siempre habrán puntos de mejora y hemos detectado al menos los siguientes.

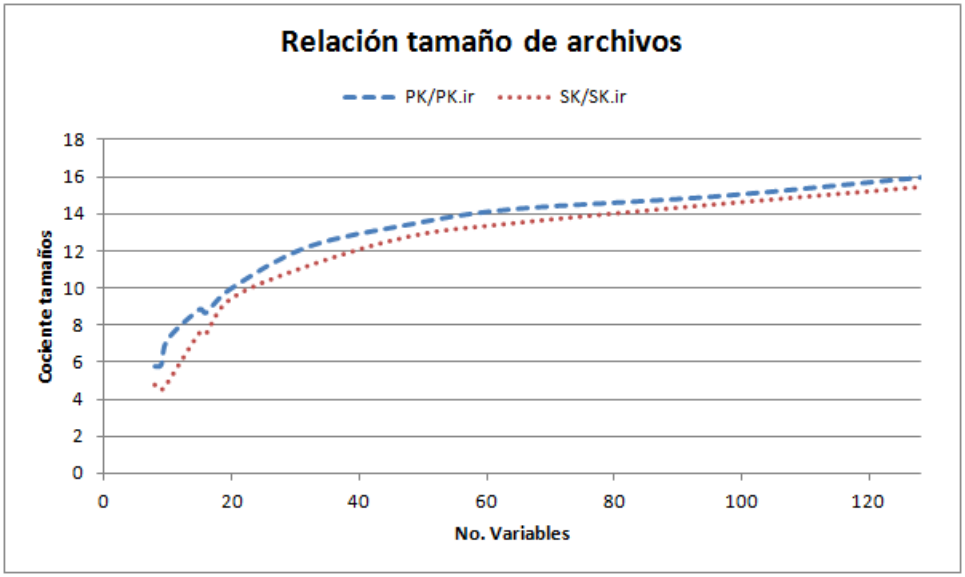

Figura 6.3: Relación representación natural vs. representación en enteros de las claves pública y privada.

6.2 Trabajo futuro

Como todo trabajo nuevo, es necesario primero probar que los conceptos están bien entendidos, que la propuesta es factible, que se ofrece una seguridad como al menos la ofrecen otros métodos actuales y si estos aspectos básicos se cumplen, hay que hacer una prueba de concepto, para asegurar que los elementos que componen la propuesta funcionan adecuadamente de forma integrada. Mucho de ésto, es lo que hemos presentado en esta tesis y ahora, como trabajo futuro, habrá que desarrollar al menos los siguientes temas, con la finalidad de poder descubrir nuevas alternativas y posibles puntos de mejora de los protocolos aquí propuestos:

I. Explicamos ampliamente como los polinomios generados se representan en lugar de su forma natural, en enteros, para que usen menos espacio en disco y la transmisión a otra entidad sea más rápida. Sin embargo, conforme el número de variables crezca el número entero que representa a los primeros términos cuadráticos y a los términos lineales, puede ser un número muy grande. Esto requiere invariablemente el uso de bibliotecas para números grandes o de múltiple precision. Es conveniente buscar otra alternativa para la representación de los polinomios, donde ocupen poco espacio y que sea de fácil manejo esta nueva alternativa.

II. Estos números grandes, en particular cuando se tienen que representar con más de 64 bits, se convierten en un problema para la codificación DER en ASN.1, para la bibliioteca que usamos (libtasn1) por lo que en vez de representar estos enteros en ASN.1 propiamente como enteros, hubo que representarlos como

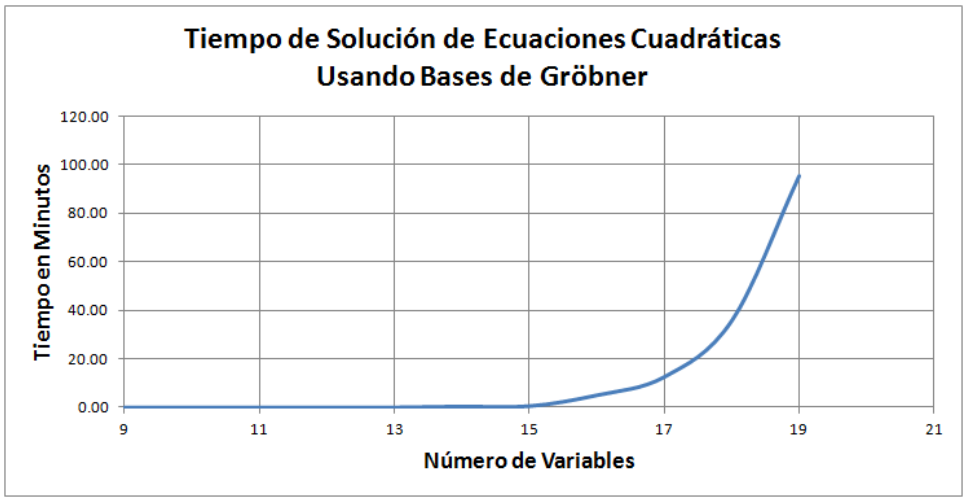

Figura 6.4: Tiempo de solución por medio de bases de Gröbner de los polinomios públicos.

cadenas de caracteres. Esto es un problema propio de la biblioteca usada, y se puede solucionar empleando otra biblioteca, sin embargo las que encontramos y que en principio no tienen este problema (por ejemplo ASN.1 de OSS-Nokalva o de Marben), no pertenecen a la comunidad de software libre.

III. El campo usado en el desarrollo completo de este trabajo fue $\mathbb{F}_2$. Sin embargo, no hay ninguna restricción para no usar algún otro campo. Se deberían realizar pruebas en campos F_q y evaluar las ventajas y desventajas de su uso, especialmente porque la dificultad para resolver los polinomios públicos por bases de Gröbner puede aumentar, dando entonces mayor seguridad a los protocolos. En el Apéndice A se muestran los programas que desarrollamos para generar claves públicas y privadas de forma más eficiente usando `C++` en campos F_q con q un número primo.

IV. En este trabajo, algunos programas fueron desarrollados en Sage y aunque existe la posibilidad de hacer una compilación de ellos para que corran de forma nativa en el sistema operativo, la eficiencia que se lograría al programar en C o C++ es mejor. Es importante entonces cambiar los programas que en Sage se interpretan, a C o C++ donde quedarían compilados de forma nativa.

V. Es importante profundizar en el manejo de las claves, para definir los detalles de como se tranferirán o se harán del conocimiento a las entidades públicas que los requieran. En este trabajo, se dan los elementos para este manejo, como son la generación de claves de forma cifrada, la generación de resúmenes (digestos, para asegurar que si cambian cuando se transfieren, ésto se pueda detectar), la codificación en DER-ASN.1 y Base 64, pero los detalles del manejo, como mencionamos en su momento, lo consideramos fuera del alcance de este trabajo.

VI. Los protocolos que se implementaron, se hicieron con el objetivo de probar que

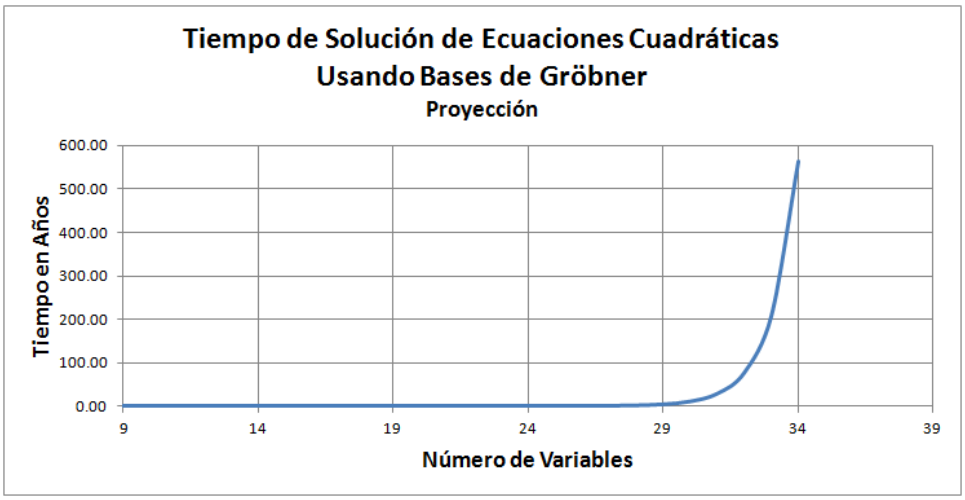

Figura 6.5: Tiempo de solución por medio de bases de Gröbner de los polinomios públicos - Estimado.

el concepto funciona y que fuera fácil dar un seguimiento a la ejecución del mismo. Es sin embargo importante desarrollarlos para su ejecución eficiente y por ejemplo, para esto hay que desarrollarlos como *demonios* a nivel del sistema operativo e incluir algunas medidas de seguridad adicionales, como lo pueden ser, números de sesión, para asegurar que un *demonio* tiene establecido correctamente el contexto en el que trabaja, en caso de varios clientes conectándose a él al mismo tiempo.

VII. Un punto final pero muy importante, es el de someter los protocolos desarrollados a una comunidad de criptoanalistas, para detectar posibles ataques y en caso que los haya, resolverlos. Si estos ataques se solucionan se podría dar el siguiente paso: presentar de forma abierta los protocolos a criptoanalistas en general, para detectar si algo más se puede encontrar y que los protocolos se vuelvan más sólidos.

Apéndice A

Generación de claves en C++

Como mencionamos en el Capítulo 5, los programas y ejemplos mostrados, se realizaron en `Sage` o alguna combinación con `Python` o `C` y en el campo F_2. Sin embargo, con la finalidad de mostrar una forma más eficiente para la generación de claves, desarrollamos los programas en `C++` para generar las claves privadas y públicas en general, en cualquier campo F_q, donde q es un número primo. Para esto, utilizamos la versión 4.9.2 del compilador `g++` para C++. Por otra parte, para realizar todas las operaciones con los polinomios, empleamos la biblioteca `GiNaC`, versión 1.6.4 que es la última versión liberada (Mayo, 2015). `GiNaC` nos permite manejar expresiones simbólicas, pero no tiene definidas operaciones en campo alguno, por lo que nuestro programa implementa la lógica necesaria para manejar campos.

En el directorio `DemoGenKeysCpp` del enlace `Programas`, se encuentran los archivos mostrados en la tabla A.1 donde también se menciona que función tiene cada uno de ellos.

Considerando que estamos ubicados bajo el directorio `DemoGenKeysCpp`, el programa ejecutable se debe correr de la siguiente manera:

```
./ModArith NoVars FieldOrd
```

donde `NoVars` representa al número de variables que se desea tengan las claves generadas y `FieldOrd` corresponde al orden del campo donde se trabajará (debe ser un número primo). Si los campos `NoVars` y `FieldOrd` no se proporcionan, entonces se generarán claves en 16 variables y en F_7. En este apéndice, consideramos que se solicitó generáran claves en 12 variables y en F_5, es decir el comando ejecutado fue:

```
./ModArith 12 5
```

Este programa, después de determinar el número de variables y el campo en que se generarán las claves, realiza la definición de las variables y estructuras de datos necesarias para trabajar. Después, genera los polinomios privados cumpliendo con la estructura AVNE y los nombra `PiUOV`. En este caso, la variables usada en estos polinomios es `a` y en la Figura A.1 se aprecian los 4 polinomios generados como

Archivo	Función
`functions.cpp`	Definición de todas las funciones en `C++`, empleadas en la generación de claves
`functions.h`	Archivo de encabezados para `C++`
`MInv.txt`	Ejemplo de la matriz inversa en la transformación afín para 12 variables en F_5
`ModArith`	Programa ejecutable para generar claves privadas y públicas
`modarith.cpp`	Archivo con el programa principal en `C++` para la generación de claves
`MsVs.txt`	Ejemplo de la matriz y vector empleados en la transformación afín, para 12 variables en F_5
`out.txt`	Salida completa del programa `ModArith` corriendo para 12 variables en F_5
`PK.txt`	Archivo de texto con la clave pública generada para 12 variables en F_5
`SK.txt`	Archivo de texto con la clave secreta generada para 12 variables en F_5

Tabla A.1: Contenido del directorio `DemoGenKeysCpp` del enlace `Programas`

clave privada. Se puede ver también el tiempo que toma esta actividad, en este caso 0.006854 segundos (todos los tiempos mostrados están en segundos).

En seguida, el programa genera los elementos para la transformación afín, es decir una matriz invertible con elementos aleatorios en el campo F_5 y un vector también en este campo. En la Figura A.2 se muestran estas matrices y el vector generados para 12 variables en el campo F_5.

Viene ahora la parte que toma más tiempo en la generación de las claves y en este caso es la generación de la clave pública. Primeramente, se genera la transformación afín, es decir la equivalencia de cada una de las variables `a` expresadas en función de las variables `x`, ver Ecuación 3.1 . Cuando se tienen estas equivalencias, será necesario evaluar los polinomios `PiUOV` en los valores encontrados y esto generará los polinomios que forman la clave pública. El resultado para el ejemplo que estamos siguiendo, se muestra en la Figura A.3. Los primeros doce valores mostrados en la figura antes mencionada, corresponden a la equivalencia de cada una de las doce variables `a`, es decir `a0` será equivalente a `4+x1+x6+3*x11+3*x3+2*x8+x10+4*x2+2*x7+3*x4+3*x9` y este valor y los once restantes que equivalen a `a1` a `a11` se deberán sustituir en las expresiones de los polinomios `PiUOV` (Figura A.1) con lo que generaremos los polinomios `Pi` (estos expresados en la variable `x`) mismos que se observan también en la Figura A.3. En esta figura, se muestra también el tiempo que tomó realizar la sustitución de las 12 variables `a` expresadas en función de `x` para generar los cuatro polinomios que forman la clave pública `Pi`, en este caso 0.047494 segundos.

Una vez que se tienen los polinomios correspondientes a las claves privadas y

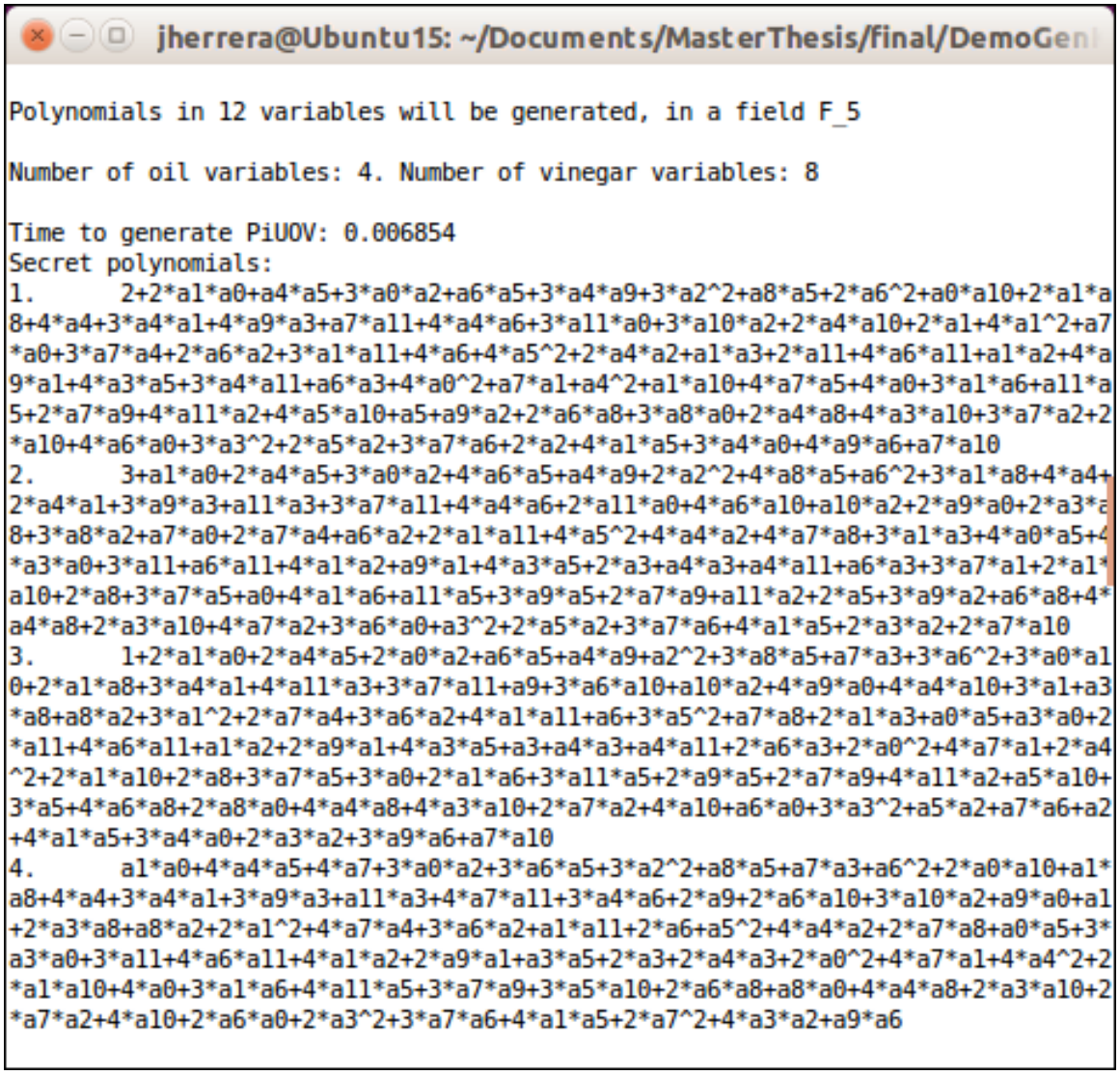

Figura A.1: Generación de polinomios de la clave secreta PiUOV para 12 variables en F_5.

públicas, el programa genera aleatoriamente una imagen `y` que deberán cumplir tanto los polinomios `PiUOV` con los valores adecuados de las variables `a`, así como los polinomios `Pi` para valores adecuados de `x`. En la Figura A.4 se muestra dicha salida deseada y los valores de las variables `a` que al sustituirse en los polinomios de la clave secreta `PiUOV`, producen esa salida.

Usando ahora la transformación inversa afín, se pueden encontrar eficientemente los valores de las variables `x` que en los polinomios `Pi` producirán la misma salida `y` antes generada. La Figura A.5 muestra este resultado.

El programa termina generando algunos archivos, que son útiles para el manejo de las claves. Como se muestra en la Tabla A.1, los archivos `PK.txt` y `SK.txt` contienen los polinomios expresados en forma natural de las claves pública y privada respectivamente. Para el caso de F_2, estos archivos pueden importarse directamente por los

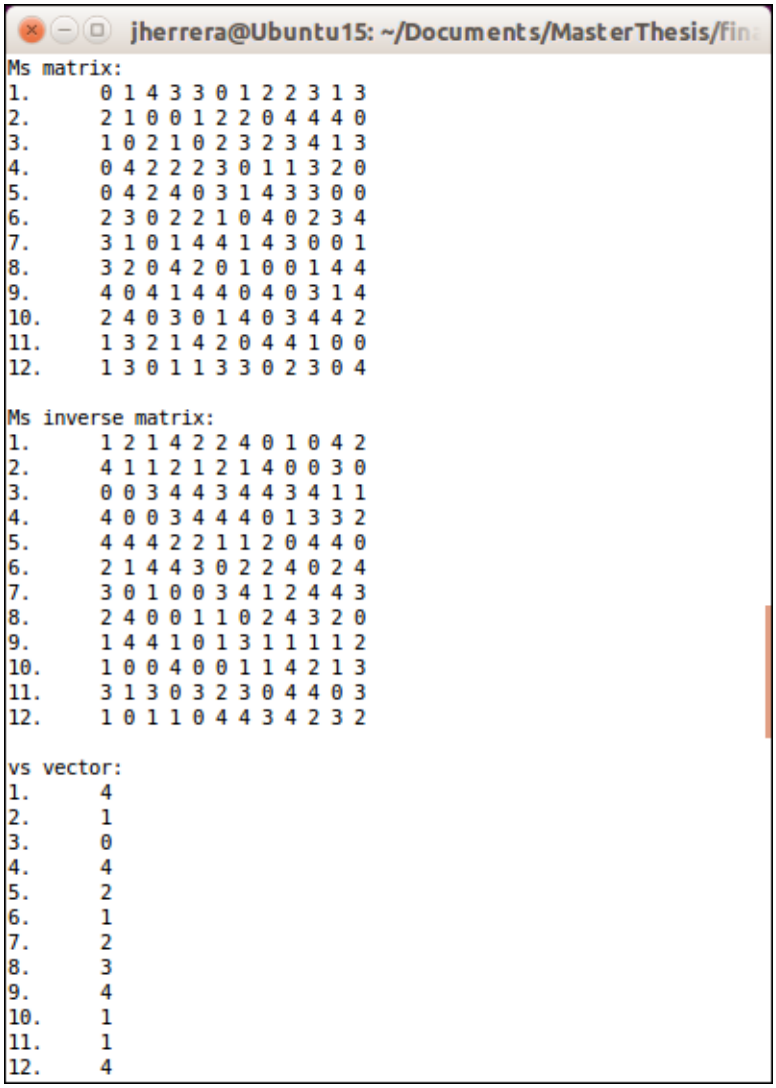

Figura A.2: Matriz, inversa de esa matriz y vector con elementos en F_5, para la transformación afín.

programas en `Sage` que se explicaron ámpliamente en el Capítulo 5. Para cualquier otro campo, se agrega al archivo generado, una primera línea, que contiene el orden del campo en que se han trabajado esos polinomios. Por otra parte, el archivo `MsVs.txt` contienen la matriz y vector de la transformación afín y el archivo `MInv.txt` tiene la matríz inversa usada por la transformación afín.

Por otra parte, en el directorio `DemoGenKeysCpp` del enlace `Programas`, se encuentra el archivo `out.txt`, que contiene la salida completa del programa `ModArith` para el ejemplo aquí mostrado: doce variables en F_5.

En la Sección 4.1.3 encontramos que en F_2, un parámetro de seguridad de 128 bits en el protocolo de autenticación, implica tener 128 polinomios en 384 variables. La ventaja de poder generar polinomios en campos mayores a F_2, es que el número

de variables se reduce dependiendo del tamaño del campo. Así, considerando un parámetro de seguridad deseado de 128 bits, q el orden del campo ($q > 2$) y ne el número de ecuaciones que se deberán generar para ese campo, tenemos:

$$\begin{aligned} q^{ne} &= 2^{128} \\ ne &= \log_q(2^{128}) \\ ne &= 128\frac{\log_{10} 2}{\log_{10} q} \end{aligned}$$

Donde $\frac{\log_{10} 2}{\log_{10} q} < 1$ y por lo tanto ne para este ejemplo siempre será menor a 128. Como un ejemplo, si queremos trabajar en F_5, el número de ecuaciones debe ser:

$$ne = 128\frac{\log_{10} 2}{\log_{10} 5} = 55.1266 \tag{A.1}$$

Es decir, tendremos que generar polinomios en $\lceil 3 \times 55.1266 \rceil = 166$ variables en F_5, contrastando con los 128 polinomios en 384 variables en F_2.

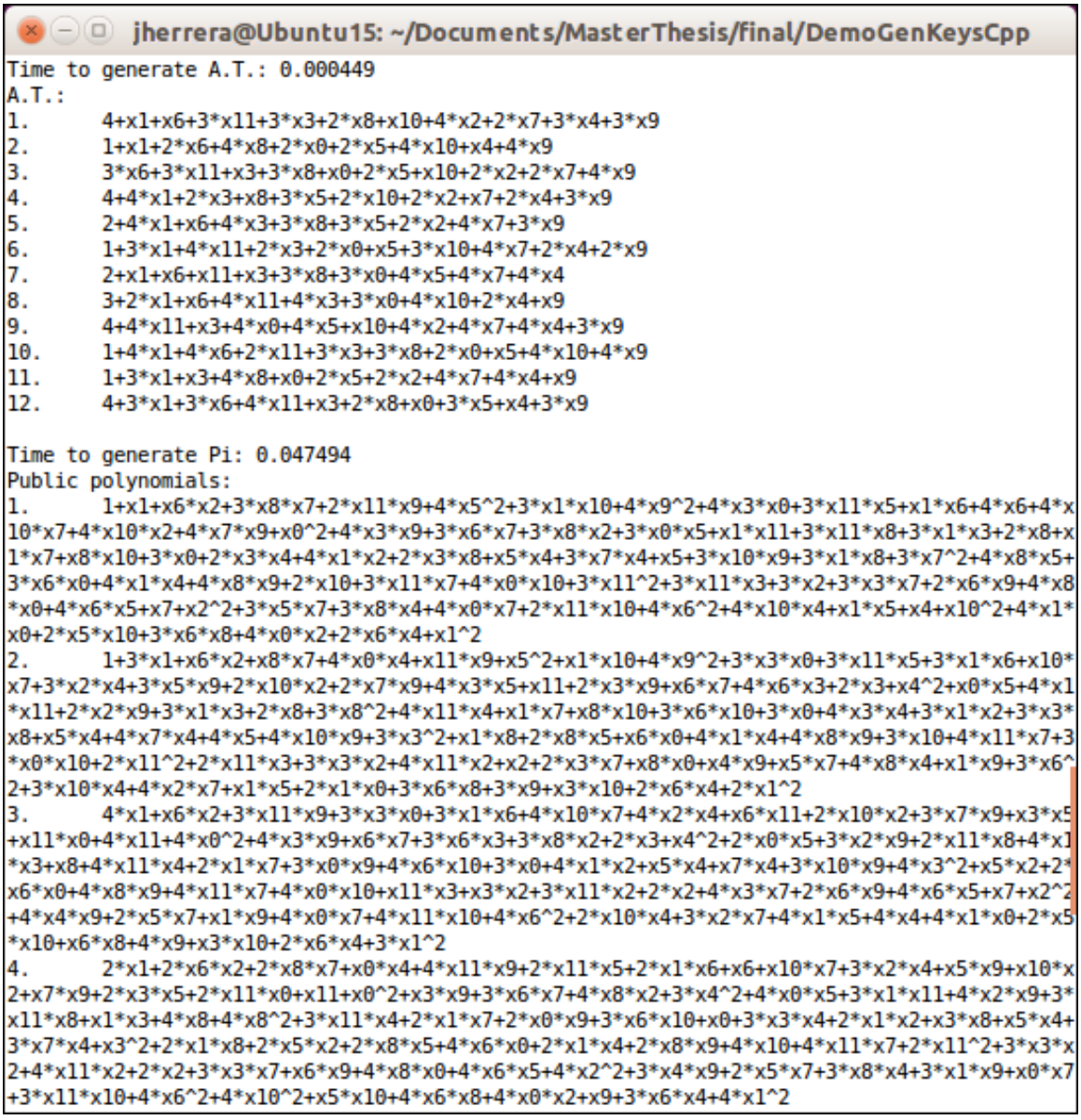

Figura A.3: Cálculo de la transformación afín, así como de los polinomios de la clave pública Pi, para 12 variables en F_5.

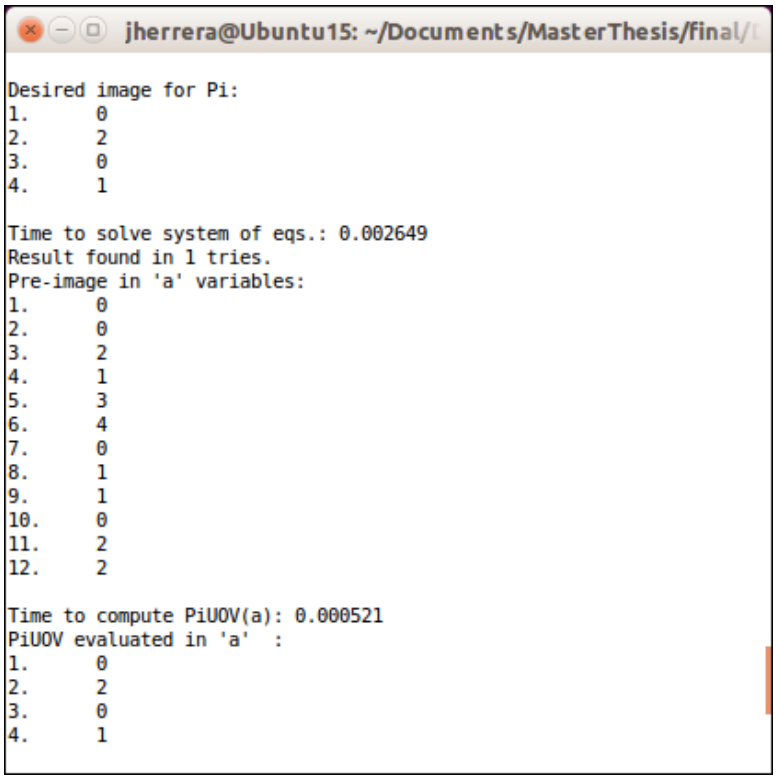

Figura A.4: Generación de una salida deseada y determinación de las variables `a` que en los polinomios secretos `PiUOV` cumplen con dicha salida.

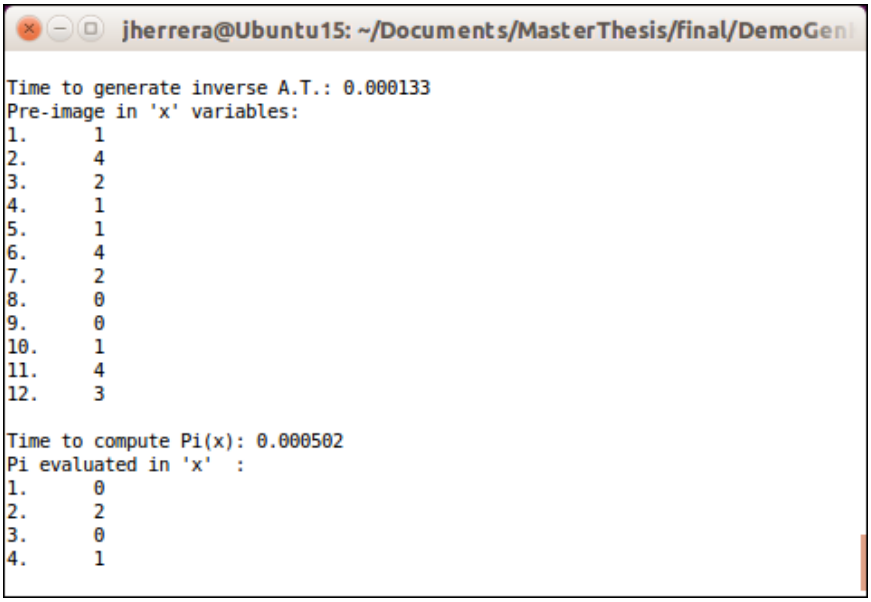

Figura A.5: Determinación de las variables `x` que al sustituirse en los polinomios públicos `Pi` cumplen con la salida deseada que se muestra en la Figura A.4.

Apéndice B

Cálculo de bases de Gröbner

Un ataque típico para el tipo de esquemas que nos ocupa en esta tesis, es el de tratar de resolver los polinomios que forman la clave pública, empleando bases de Gröbner, que nos entregará una serie de polinomios en donde haciendo una sustitución de variables hacia adelante[1] se podrán encontrar los valores de las n variables involucradas en los polinomios. Como dijimos antes, esto será posible si es que podemos encontrar una base de Gröbner para los polinomios de la clave pública.

En el directorio `GroebnerBasis` del enlace `Programas`, se encuentran los archivos mostrados en la tabla B.1 en donde también se menciona que función tiene cada uno de ellos.

El programa `gbParallel.py` o `gb.py`, realizan las funciones siguientes, en lo único en que difieren, es que el primero explota el paralelismo que ofrece un equipo con varios cores, mientras que el segundo es para equipos que sólo tienen un core disponible:

1. Primeramente, pide el nombre del archivo que contiene los polinomios que generan el ideal a calcular su base de Gröbner. Después pide el nombre del archivo donde se guardará la base de Gröbner que calcule y finalmente el número de veces que buscará dicha base, esto con el fin de poder calcular el promedio que tarda encontrar esta base.

2. Lee ahora el archivo con los polinomios que generan el ideal y los guarda en un archivo temporal llamado `forPB`, con el comando que declara un anillo para el número de variables en los polinomios y con la sintaxis adecuada que entiende PolyBori de un polinomio booleano.

3. Una vez generado el archivo `forPB` lo lee, generando una estructura de datos adecuada para el manejo en PolyBori.

4. Con esta estructura de datos ya en memoria, realiza el cálculo de la base de Gröbner tantas veces como se le haya solicitado al inicio del programa, tomando

[1] Es decir, tomar el primer polinomio de la base de Gröbner generada y sustituir una variable por un valor aleatorio (cero o uno, ya que estamos hablando de polinomios booleanos) y con base en esto ir determinando los valores de las variables que queden, barriendo todos los polinomios de la base, hasta encontrar los valores de las n variables.

Archivo	Función
`comoEjecutar`	Archivo de texto, con instrucciones de como generar las bases de Gröbner
`errorPK20.txt`	Mensaje de error generado por PolyBoRi para polinomios de 20 variables
`forPB`	Archivo temporal que genera el programa para el cálculo de las bases de Gröbner
`gbParallel.py`	Programa para obtener las bases de Gröbner en un servidor multicore
`gb.py`	Programa para obtener las bases de Gröbner sin usar varios cores
`PK*`	Archivos `PK9,...,PK19` con polinomios en su forma natural, en `9,...,19` variables
`PK*.gb`	Archivos `PK9.gb,...,PK19.gb` con la base de Gröbner generada por PolyBori usando multicores y el tiempo transcurrido para obtenerla
`data/`	Directorio con archivos de polinomios en su forma natural y los generados por PolyBori con bases de Gröbner, sin usar varios cores
`examples/`	Directorio con archivos ejemplos simples

Tabla B.1: Contenido del directorio `GroebnerBasis` del enlace `Programas`

el tiempo para calcular esta base. Al terminar todo el ciclo, guarda en el archivo cuyo nombre se pidió al inicio de este programa la base que encontró, junto con el tiempo promedio que tomo la generación de esta base.

5. Finalmente, y usando la base de Gröbner que se acaba de encontrar, busca los valores de las variables de los polinomios, proponiendo un valor para la primera variable del primer polinomio de la base y observando que mas se puede sustituir recursivamente en los siguientes polinomios, hasta agotar todos, registrando los valores propuestos y los que se vayan determinando al reducir los polinomios.

6. Imprime los valores de las variables encontradas.

Si se tiene acceso a un computador con varios cores, se debe ejecutar el programa `gbParallel.py`, ya que éste explota la generación de la base de Gröbner en paralelo. Sin embargo, si sólo se tiene acceso a computadores con un sólo core, se puede correr el programa `gb.py` que generará la base de Groebner de forma secuencial. Los archivos `PK*.gb` que se encuentran directamente bajo el directorio `GroebnerBasis` contienen la base de Gröbner para los polinomios en el respectivo archivo `PK*`[2] y en este caso son el producto del programa `gbParallel.gb` que se ejecutó en un servidor Linux

[2]Por ejemplo, el archivo `PK9.gb` contiene la base de Gröbner del archivo `PK9`.

con 4 cores y 64 GB de memoria. Considerando que se tiene acceso a un equipo Linux, con PolyBori installado (incluyendo la interface `ipbori`), ejecutar los siguientes comandos, para generar la base de Gröbner y encontrar el valor de las n variables de los polinomios en el correspondiente archivo `PK*`:

`ipbori`	(Levanta ambiente de PolyBoRi)
`run gb.py`	(Corre programa para generar base de Gröbner)

Con esto, el programa `gb.py` se ejecutará y pedirá primero el nombre del archivo que contiene los polinomios expresados en su forma natural, por ejemplo se puede dar `PK9` que corresponde a un archivo con 3 polinomios en 9 variables. Posteriormente pedirá el nombre del archivo donde se almacenará el resultado de la base de Gröbner junto con el tiempo promedio que tomó este cálculo, dar por ejemplo `PK9.gb`. Pedirá finalmente el número de veces que repetirá estos cálculos con la finalidad de obtener un promedio del tiempo que toma el cálculo de esta base de Gröbner. Inicia entonces con el cálculo de la base de Gröbner y al encontrar ésta, determina los valores de las variables de los polinomios, mismos que se muestran como salida del programa.

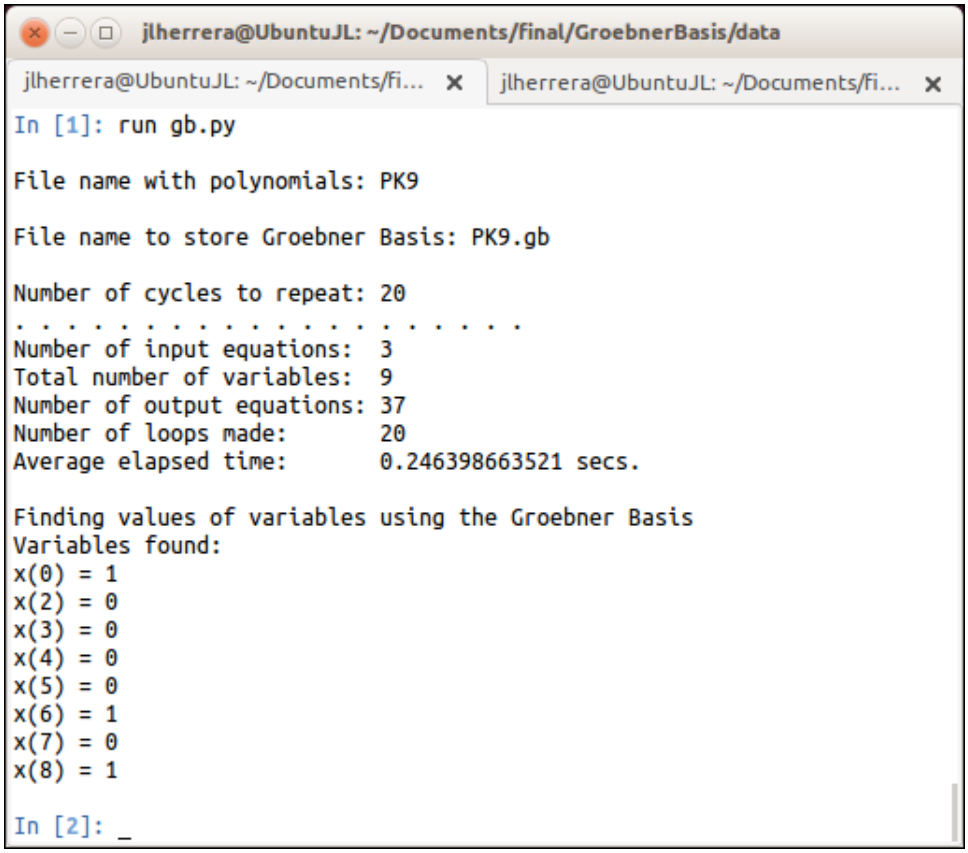

Figura B.1: Generación de base de Gröbner para tres polinomios en nueve variables: PK9

En la Figura B.1 se muestran los pasos anteriores y podemos ver que de las 3 ecuaciones que se dieron al programa, se generaron 37 ecuaciones con la base de Gröbner y

el tiempo que tomo en promedio el calcular dicha base, fue de 0.24 segundos. Al final de esta figura, vemos el valor que encontro para $\mathbf{x} = x_0, \ldots, x_8 = 1, x_1, 0, 0, 0, 0, 1, 0, 1$, donde sólo el valor para x_1 no quedo completamente determinado lo que significa que puede tomar cualquier valor.

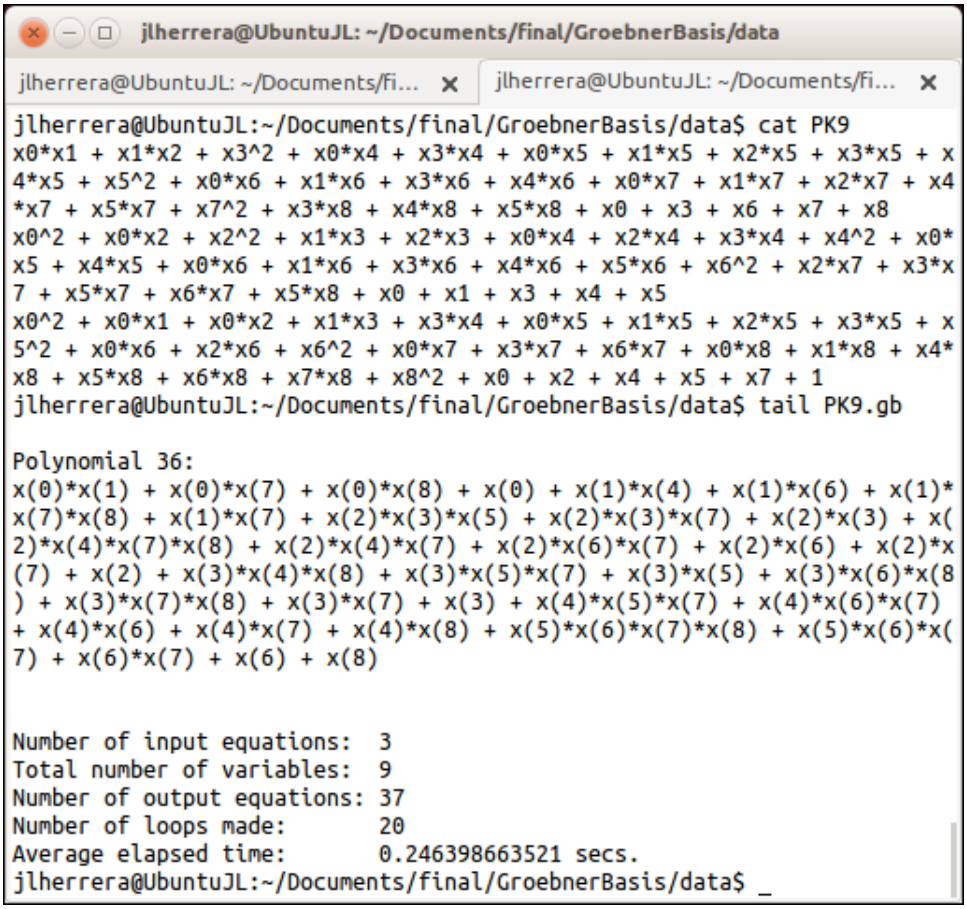

Figura B.2: Contenido del archivo con 3 polinomios en 9 variables `PK9` y parte final del archivo con la base de Gröbner `PK9.gb`.

La Figura B.2 muestra en su primera mitad, el contenido del archivo `PK9` es decir una clave pública en 9 variables ($n = 9$) y por lo tanto, 3 ecuaciones ($m = o = \lfloor n/3 \rfloor$). En la segunda mitad, se puede apreciar la parte final del archivo que se generó con la base de Gröbner, en este caso `PK9.gb`. Los polinomios generados se enumeran del 0 al 36 y es este último el que aparece en esta figura. Al final, vemos que el tiempo promedio que tomó la generación de la base de Gröbner, fue de 0.2463 segundos, obteniendo este promedio de las 20 veces que se solicitó se realizara el cálculo de dicha base.

En el directorio `GroebnerBasis` y como se describe en la tabla B.1 se tienen los resultados del cálculo de bases de Gröbner desde 9 hasta 19 variables. Cuando se intentó realizar para 20 variables, se obtuvo el mensaje de error que se tiene en el archivo `errorPK20.txt` y que básicamente es este: `ValueError: Built-in matrix-size`

`exceeded!`. Para 21 variables se obtiene el mismo error y por las pruebas que realizamos, llegamos a la conclusión, que la memoria que puede manejar PolyBoRi para cuando el polinomio tiene un número mayor o igual a 20 variables no es suficiente para seguir con el procesos de cálculo de la base y por lo tanto otras alternativas u otros programas se deben buscar para encontrar los valores de la imagen inversa que haga que $\mathcal{P} : x \mapsto y$.

En la sección 6.1 se presentan datos adicionales de los puntos antes expuestos, y se ve que el comportamiento en tiempo para el cálculo de las base de Gröbner usando PolyBoRi, es exponencial. No tenemos los datos exactos del uso de memoria para estos mismos casos, pero para el caso de 20 variables, lo que determinamos es que PolyBoRi estaba usando 10 GB de los 64 GB disponibles en el servidor, sin embargo, estos 10 GB no fueron suficientes para que siguiera con el proceso de encontrar la base.

Apéndice C

Acrónimos y abreviaciones

Símbolo	**Descripción**
3-SAT	SAT restringido a claúsulas de tres literales
ASN.1	Abstract Syntax Notation 1
AVNE	Aceite-Vinagre No-Equilibrado
Base64	Codificación en base 64
Bash	Bourne again shell
CAS	Computer Algebra System
DH	Diffie-Hellman. Establecimiento de claves
ECC	Elliptic Curve Cryptography
ElGamal	Procedimiento de cifrado/descifrado
$\mathbb{F}_2$	Campo de dos elementos: 0,1
GiNaC	**G**iNaC **i**s **N**ot **a** **C**AS
HFE	Hidden Field Equations
IND-CCA	Indistinguishable Chosen CypherText Attack
.ir	Extension de un archivo con representación en enteros de un polinomio (integer representation)
MIA	Matsumoto-Imai esquema A
MQ	Multivariate Quadratic Equations
MQQ	Multivariate Quadratic Quasigroups
.mv	Extension de un archivo con representación en enteros de una matriz y un vector
NP	Problemas comprobables en tiempo polinomial
NP-Completo	Problema en NP y todo problema en NP se reduce a él
OV	Esquema Oil-Vinegar
$\mathcal{P}$	Representa el conjunto de polinomios de la llave pública
$\mathcal{P}'$	Representa el conjunto de polinomios de la llave privada
PolyBoRi	Polynomials over Boolean Rings: Infraestructura para calculos en anillos booleanos
RSA	Rivest, Shamir y Adleman. Sistema criptográfico de clave pública
Sage	Sistema de software para matemáticas

Símbolo	Descripción
SAT	Problema de decisión de satisfactibilidad booleana
STS	Stepwise Triangular Systems
UOV	Esquema Unbalanced Oil and Vinegar

Bibliografía

Adams, W. W. and Loustaunau, P. *An Introduction to Gröbner Bases*. American Mathematical Society, Rhode Island, USA, 1st edition, 1994.

Bardlet, M. On the Complexity of a Gröbner Basis Algorithm. In *Algorithms Seminar*. Chyzak, F., editor, pages 85–92, France, November 25, 2002. INRIA.

Barreto, P., Piazza, F., Dahab, R., et al. A Panorama of Post-quantum Cryptography. In *Open Problems in Mathematics and Computational Science*. Çetin Kaya Koç, editor, volume 1, pages 404–411, Istanbul, Turkey, September 18-20, 2013. Springer.

Buchberger, B. Gröbner Bases: A Short Introduction for System Theorists. In *Computer Aided Systems Theory - EUROCAST 2001*. Moreno-Díaz, R., Buchberger, B., and Freire, J. L., editors, volume 2178, pages 1–19, Las Palmas de Gran Canaria, Spain, February 19-23, 2001. Springer.

Courtois, N., Klimov, A., Patarin, J., and Shamir, A. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Advances in Cryptology - EUROCRYPT 2000*. Preneel, B., editor, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407, Bruges, Belgium, May 14-18, 2000. Springer Berlin Heidelberg.

Cox, D., Little, J., and O'Shea, D. *Ideals, Varieties and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, MA, USA, 4th edition, 2015.

Diffie, W. and Hellman, M. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November, 1976.

Ding, J., Petzoldt, A., and chung Wang, L. The Cubic Simple Matrix Encryption Scheme. In *Post-Quantum Cryptography*. Mosca, M., editor, volume 8772, pages 76–87, Waterloo, ON, Canada, October 1-3, 2014. Springer.

Dubuisson, O. *ASN.1 Communication between Heterogeneous Systems*. OSS Nokalva, USA, 1st edition, 2000.

Faugère, J. C. A new efficient algorithm for computing Gröbner bases (F4). *Journal of Pure and Applied Algebra*, 139(1-3):61–88, June, 1999.

Faugère, J. C. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, ISSAC '02, pages 75–83, New York, NY, USA, 2002. ACM.

Gao, S., IV, F. V., and Wang, M. A new algorithm for computing Gröbner bases. *Cryptology ePrint Archive, Report 2010/641*, 2010(641):1–19, December, 2010.

Garey, M. R. and Johnson, D. S. *Computers and Intractability, A Guide to the Theory of NP-Completness*. W. H. Freeman & Co., New York, NY, USA, 1st edition, 1990. ISBN 0716710455.

Gligoroski, D. and Samardjiska, S. The Multivariate Probabilistic Encryption Scheme MQQ-ENC. *Cryptology ePrint Archive, Report 2012/328*, 2012(328), 2012.

Gligoroski, D., Markovski, S., and Knapskog, S. J. Multivariate Quadratic Trapdoor Functions Based on Multivariate Quadratic Quasigroups. In *Proceedings of the American Conference on Applied Mathematics*. University, C. L., Sohrab, S. H., Bognar, G., and Perlovsky, L., editors, MATH'08, pages 44–49, Stevens Point, Wisconsin, USA, March 24-26, 2008. World Scientific and Engineering Academy and Society (WSEAS). ISBN 978-960-6766-47-3. URL http://dl.acm.org/citation.cfm?id=1415583.1415596.

Goldreich, O. *Foundations of Cryptography. Basic Tools*. Cambridge University Press, Cambridge, UK, 1st edition, 2004.

Goldreich, O. *Computational Complexity. A Conceptual Perspective*. Cambridge University Press, Cambridge, UK, 1st edition, 2008.

Kipnis, A. and Shamir, A. Cryptanalysis of the oil and vinegar signature scheme. In *Advances in Cryptology - CRYPTO '98*. Krawczyk, H., editor, volume 1462, pages 257–266, Santa Barbara, Cal., USA, August 23-27, 1998. Springer.

Kipnis, A., Patarin, J., and Goubin, L. Unbalanced Oil and Vinegar Signatures Schemes. In *Advances in Cryptology - EUROCRYPT'99*. Stern, J., editor, volume 1592, pages 206–222, Prague, Czech Republic, May 2-6, 1999. Springer.

Menezes, A. J., Vanstone, S. A., and Oorschot, P. C. V. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996. ISBN 0849385237.

Nachef, V., Patarin, J., and Volte, E. Zero-knowledge for multivariate polynomials. In *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*. Hevia, A. and Neven, G., editors, pages 194–213, Santiago, Chile, October 7-10, 2012. doi: 10.1007/978-3-642-33481-8_11. URL http://dx.doi.org/10.1007/978-3-642-33481-8_11.

Patarin, J. and Goubin, L. Trapdoor One-Way Permutations and Multivariate Polynomials. In *Proc. of ICICS'97, LNCS 1334*, pages 356–368. Springer, 1997.

Porras, J., Baena, J., and Ding, J. ZHFE, a New Multivariate Public Key Encryption Scheme. In *Post-Quantum Cryptography*. Mosca, M., editor, volume 8772, pages 219–245, Waterloo, ON, Canada, October 1-3, 2014. Springer.

Rivest, R., Shamir, A., and Adleman, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February, 1978.

Sakumoto, K., Shirai, T., and Hiwatari, H. Public-Key Identification Schemes Based on Multivariate Quadratic Polynomials. In *Advances in Cryptology - CRYPTO 2011*. Rogaway, P., editor, volume 6841, pages 706–723, Santa Barbara, CA, USA, August 14-18, 2011. Springer.

Shor, P. W. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. on Computing*, 26(5):1484–1509, October, 1997.

Tao, C., Diene, A., Tang, S., and Ding, J. Simple Matrix Scheme for Encryption. In *Post-Quantum Cryptography*. Gaborit, P., editor, volume 7932, pages 231–242, Limoges, France, June 4-7, 2013. Springer.

Wolf, C. and Preneel, B. Taxonomy of Public Key Schemes based on the problem of Multivariate Quadratic equations. *Cryptology ePrint Archive, Report 2005/077*, 2005(077):4–54, December, 2005a.

Wolf, C. and Preneel, B. A Study of the Security of Unbalanced Oil and Vinegar Signature Schemes. In *Topics in Cryptology - CT-RSA 2005*. Menezes, A., editor, volume 3376 of *Lecture Notes in Computer Science*, pages 29–43, San Francisco, CA, USA, February 14-18, 2005b. Springer Berlin Heidelberg.

Wolf, C. *Multivariate Quadratic Polynomials in Public Key Cryptography*. PhD thesis, Katholieke Universiteit Leuven, Departement Elektrotechniek-Esat, 2005.