UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SISTEMA DE CRIPTOANÁLISIS DISTRIBUIDO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

EDUARDO EMMANUEL ALEXIS FRÍAS MORALES

PROFESOR GUÍA:
ALEJANDRO HEVIA ANGÚLO

MIEMBROS DE LA COMISIÓN:
GONZALO NAVARRO BADINO
PABLO GUERRERO PÉREZ

SANTIAGO DE CHILE
2014

Resumen

Un supuesto criptográfico es un problema computacional que se conjetura difícil de resolver con algoritmos que utilizan tiempo y recursos razonables, para los adversarios considerados. Ejemplos de esto son: factorización y encontrar el logaritmo discreto. Estos supuestos permiten definir esquemas criptográficos seguros si los supuestos son verdaderos. En relación a esto, el uso de técnicas tanto computacionales como matemáticas para intentar resolver los problemas computacionales subyacentes a dichos supuestos, se denomina criptoanálisis.

Existen diversos algoritmos criptoanalíticos que intentan resolver supuestos computacionales, tanto de forma secuencial como distribuida. Estos algoritmos deben ser implementados o adaptados por una persona interesada en el tema, ya sea para realizar un análisis sobre éstos, resolver un supuesto construido sobre otro(s), o para algún otro uso relevante para un criptógrafo. Sin embargo, no siempre se tiene a disposición el hardware adecuado para este tipo de trabajos: ejecutar un algoritmo criptoanalítico sobre entradas de muchos bits (el caso típicamente interesante) en un computador de escritorio, puede resultar infactible. Por otro lado, si es que se tuviera, por ejemplo, un cluster a disposición del investigador, el acceso a éste podría ser complejo y poco adecuado, dificultando el trabajo a realizar. Es por esto que se intenta buscar una solución que permita al criptógrafo enfocarse en lo importante de la investigación, que permita implementar un algoritmo distribuido de manera simple, modular e independiente del hardware.

Este trabajo se separó en dos partes: desarrollo de una plataforma web, y configuración de un cluster en Amazon Web Services. En la plataforma web se implementó: un login que permitiese autentificar a los usuarios, en donde un administrador puede crearlos; un sistema de resolución de problemas computacionales; una manera sencilla para agregar otros problemas; un mecanismo para que un usuario pueda implementar algoritmos personalizados; y un sistema que permite ver el estado de las tareas. Con respecto a la configuración del cluster, se implementaron algoritmos distribuidos para resolver algunos de los problemas computacionales más conocidos, los cuales permiten: factorizar un número; encontrar el logaritmo discreto en el grupo $\mathbb{Z}_p^*$, y en el grupo de las curvas elípticas sobre $\mathbb{F}_p$, donde p es primo; encontrar una colisión y preimagen de una función de hash.

Con esta solución se llegó a una aplicación que alcanzó los objetivos propuestos, pues un usuario tiene un acceso relativamente fácil al sistema (aunque filtrado por un encargado), puede ejecutar de forma inmediata algoritmos criptoanalíticos sin tener que implementarlos y puede ejecutar algoritmos criptoanalíticos personalizados que le otorgan una mayor flexibilidad sobre las operaciones disponibles.

Agradecimientos

Le doy las gracias a mi profesor guía, Alejandro Hevia, por todo el apoyo y la ayuda entregada en este trabajo de título.

También le doy las gracias a mi familia y amigos, que estuvieron brindándome su apoyo en todo momento, antes y durante este trabajo de título.

Tabla de Contenido

Resumen I

Tabla de Contenido III

1. Introducción 1

1.1. Puntos Claves 2

1.2. Alternativas 2

1.3. Justificación 3

1.4. Descripción general de la solución 3

2. Antecedentes 5

2.1. Conceptos principales 5

2.1.1. Grupos Abelianos 5

2.1.1.1. Grupo de los enteros módulo n bajo la multiplicación ($\mathbb{Z}_n^*$) y suma ($\mathbb{Z}_n$) 7

2.1.1.2. Grupo de las curvas elípticas 8

2.1.2. Criptografía 10

2.1.3. Supuestos computacionales 10

2.1.4. Criptoanálisis 11

2.2. Arquitecturas Estándares 12

2.2.1. Amazon Web Services 12

2.2.1.1. Amazon EC2 13

2.3. Herramientas útiles 15
2.3.1. Message Passing Interface (MPI) 15
2.3.1.1. Conceptos principales 15
2.3.2. Play! Framework 17
2.4. Algoritmos considerados 17
2.4.1. Factorización 17
2.4.1.1. Trial Division 18
2.4.1.2. Pollard's rho 18
2.4.1.3. Pollard's $p-1$ 19
2.4.1.4. Lenstra's Elliptic Curve Method 20
2.4.1.5. Factorización por diferencia de cuadrados 21
2.4.1.6. Quadratic Sieve 22
2.4.1.7. Number Field Sieve 23
2.4.2. Logaritmo Discreto 24
2.4.2.1. Fuerza Bruta 24
2.4.2.2. Shanks' Algorithm 25
2.4.2.3. Pohlig-Hellman Algorithm 26
2.4.2.4. Pollard's rho 27
2.4.2.5. Index Calculus 28
2.4.3. Colisiones de Hash 30
2.4.3.1. Fuerza bruta 30
2.4.3.2. Pollard's rho 30
2.4.4. Preimágenes de Hash 31
2.4.4.1. Fuerza bruta 32
2.5. Soluciones existentes que realizan criptoanálisis 32
2.5.1. Máquinas personalizadas (Custom hardware attack) 32
2.5.2. Volunteer Computing 34
2.5.3. Otros 35

3. Especificación del Problema **37**

3.1. Descripción del problema a resolver 37

3.2. Relevancia de contar con una solución 38

3.3. Requisitos de la solución 38

4. Descripción de la Solución **40**

4.1. Arquitectura del hardware 40

4.1.1. Servidor Frontend 40

4.1.2. Cluster AWS 40

4.2. Arquitectura del software 41

4.2.1. Servidor Frontend 41

4.2.2. Cluster AWS 41

4.3. Diseño de la base de datos 42

4.3.1. Tabla Usuarios 42

4.3.2. Tabla Estados 42

4.4. Diseño de la aplicación 43

4.4.1. Servidor Frontend 43

4.4.1.1. Controllers 43

4.4.1.2. Models 46

4.4.1.3. Views 48

4.4.1.4. Util 50

4.4.1.5. Supuestos.base 53

4.4.1.6. Supuestos 56

4.4.2. Cluster AWS 58

4.4.3. Uso de los scripts de Usuario 61

4.4.4. Extensibilidad de Supuestos 62

4.5. Diseño de la Interfaz de Usuario 63

4.6. Argumentación de la solución 64

5. Validación de la Solución **65**

6. Conclusiones **70**

6.1. Trabajo futuro 71

Bibliografía **74**

Capítulo 1

Introducción

La comunidad académica de la facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile no posee una forma sencilla de realizar criptoanálisis de forma rápida sobre los variados problemas computacionales que existen, y de los cuales muchos dependen de otros.

Informalmente hablando, el criptoanálisis es el empleo de diversas técnicas matemáticas y algorítmicas que permiten resolver un problema computacional, como lo es, por ejemplo, el problema de la factorización, el cual es considerado difícil. Es por esto, que se han desarrollado una serie de algoritmos que permiten resolver varios de estos problemas computacionales. En la mayoría de los casos, las implementaciones a estos algoritmos se encuentran disponibles como código OpenSource para descargar. Sin embargo, su instalación puede no ser sencilla dado que, en general, no suelen ser códigos 100 % portables, y típicamente dependen de librerías poco estándares, las cuales rara vez se encuentran instaladas en el sistema. En cuanto a su uso, muchas implementaciones son poco claras, complejas de usar, y tienen una documentación poco adecuada e incompleta. Por otro lado, no siempre se tiene el poder computacional necesario para ejecutar las aplicaciones criptoanalíticas, las cuales requieren recursos computacionales altos. Ejecutarlas en un computador de uso común podría tomar varios años o incluso siglos.

Se nota entonces que un método sencillo para realizar implementaciones de ataques criptoanalíticos es necesario, pues si bien existen diversas técnicas y herramientas teóricas que sirven para aproximar el tiempo que toman los algoritmos, en la práctica no siempre son precisas, pues dependen fuertemente de cómo y sobre qué arquitectura estos ataques teóricos son implementados. Por ejemplo, Deep Crack, una máquina construida en 1998 especialmente para realizar un ataque de fuerza bruta sobre DES [33] (i.e., probar todas las posibles llaves), fue capaz de encontrar la llave secreta en menos de 56 horas, lo cual hubiese tardado cientos de años con los computadores usuales de aquella época. Es por esto, que si existiera una forma sencilla de realizar tales implementaciones, se podría medir de forma efectiva el tiempo que tardan, bajo una implementación y arquitectura bien definida.

1.1. Puntos Claves

Con esto en mente, se definió que la solución a este problema tenía que cumplir ciertos puntos fundamentales para que fuese considerada una forma ’sencilla’ de realizar criptoanálisis. Primero, esta forma o sistema tiene que ser de fácil acceso a la comunidad académica. En segundo lugar, tiene que ser escalable, en el sentido de que se pueda agregar un nuevo ataque criptoanalítico al sistema en pocos pasos. Tercero, que sea de fácil uso, para que se pueda instanciar un nuevo algoritmo criptoanalítico de forma sencilla, e incluso que tenga la capacidad de que un usuario pueda escribir su propio ataque en base a los ya implementados en el sistema. En cuarto y último lugar, que sea eficiente, permitiendo completar estos ataques en tiempos mucho menores a los usados por un computador de escritorio.

1.2. Alternativas

Estos puntos claves permiten la elección de una alternativa entre varias posibles. En cuanto a la eficiencia del sistema, primero se tiene que obtener un hardware adecuado y al alcance de este trabajo. Con esto en mente, se tenían las siguientes tres alternativas: Cluster de procesadores, “volunteer computing” [1], máquina “personalizada”. La máquina personalizada fue descartada, debido a que la cantidad de dinero usada para la compra o construcción de ésta, estaba totalmente fuera del alcance de este trabajo. Un sistema basado en volunteer computing no requiere hardware personalizado, sin embargo, hay que conseguir una buena cantidad de voluntarios para que se tenga un poder computacional razonable, y aún así, puede no ser confiable debido a la variabilidad en la cual los usuarios prestan su poder de cómputo, por lo cual fue finalmente descartado. Un cluster de procesadores se encuentra disponible en la facultad, a cargo del Centro de Modelamiento Matemático (CMM) de la Facultad de Ciencias Físicas y Matemáticas, sin embargo, su acceso no es necesariamente fácil. Por esto, se escogió realizar un cluster usando máquinas virtuales provistas por Amazon Web Services (AWS) que tienen un costo asociado por hora de uso, pero son de fácil acceso, lo cual es suficiente para esta etapa del trabajo.

En cuanto a la accesibilidad del sistema, se escogió realizar una página web. Otras alternativas, como un acceso por consola o una aplicación gráfica personalizada se descartaron debido a que implementar un servidor y una aplicación gráfica desde cero, no es una tarea trivial y es más complejo hacerlo multiplataforma. Por otro lado, la interfaz con el usuario en una página web es mucho más simple de implementar. La facilidad de uso también puede ser obtenida con una página web de manera mucho más sencilla, debido a la facilidad de introducir elementos que interactúan con los usuarios.

Finalmente, la escabilidad está dada por la calidad del código, sin embargo, hay lenguajes que facilitan el realizar aplicaciones escalables, como por ejemplo Scala, el cual se escogió como lenguaje para realizar los procesos importantes de la plataforma web, y con el cual será posible la inclusión de nuevos ataques de manera simple.

1.3. Justificación

La dificultad de este trabajo radica en una variedad de puntos. En primer lugar, la organización de un sistema distribuido de máquinas virtuales o cluster, de forma que sea posible agregar nuevas máquinas virtuales al cluster, siendo tan fácil como crear una nueva instancia o máquina virtual en AWS. Por otro lado, hay que manejar la paralelización en los distintos cores del cluster de los ataques criptoanalíticos a implementar, lo cual no es un trabajo sencillo. En tercer lugar, se tienen que desarrollar los procesos de negocio relevantes para el sistema, como por ejemplo la ejecución, desde una página web, de un algoritmo criptoanalítico implementado en el cluster. En este mismo punto, se tiene que soportar que código escrito por el usuario pueda ser ejecutado, cuidando de lo que puede y no puede hacer, y así lograr un sistema personalizable. También se tiene que asegurar que el acceso a este sistema sea exclusivo para usuarios autorizados y que no cualquier persona pueda acceder a él. Por último, se tiene que resolver el problema de la utilización de los recursos de las tareas iniciadas por los usuarios, para que dos tareas distintas no usen el mismo core o núcleo del cluster, pues esto deterioraría el rendimiento de dichas tareas.

1.4. Descripción general de la solución

Finalmente, se listarán los pasos principales que tiene la solución propuesta a este trabajo:

- Desarrollo de una interfaz web que permita:
  - El registro de los usuarios, en la cual el/los administradores tienen las facultades para aceptar o no a los usuarios registrados. Esto para evitar usuarios desconocidos/malintencionados.
  - A los usuarios la ejecución de ataques, en donde el back-end se encuentre en el cluster.
  - A los usuarios la ejecución de ataques personalizados, que dependen de ataques criptoanalíticos ya introducidos en el sistema.
  - A los usuarios ver el estado de las tareas ejecutadas, ya sea tareas completas, canceladas o en ejecución.
  - A un programador la capacidad de introducir al sistema nuevos ataques criptoanalíticos, de forma de que los usuarios puedan usar estos nuevos ataques fácilmente. Estos ataques pueden o no depender de los ya introducidos en el sistema. Por ejemplo, se podría agregar al sistema la capacidad de que los usuarios puedan quebrar RSA [33] o que puedan factorizar usando otro algoritmo.
- Creación de un cluster

    - Que tenga todas las librerías necesarias para el uso de los algoritmos criptoanalíticos.
    - Que tenga un tamaño dinámico, es decir, que la cantidad de núcleos disponibles dependa de la cantidad de instancias iniciadas en AWS.

- Implementación de ataques criptoanalíticos distribuidos en el cluster para el uso de la plataforma web:

    - Que usen una cantidad de núcleos que dependa del input del usuario.
    - Que el programa escoja un algoritmo específico, dependiendo de la entrada (por ejemplo, para factorizar, elegir Trial Division para números pequeños, y otros algoritmos para números más grandes)

Esta memoria está dedicada al desarrollo de un sistema que se encargue de ejecutar y aplicar distintos algoritmos de criptoanálisis para atacar, de forma práctica, problemas de teoría de números o combinatorios presentes en sistemas criptográficos.

La estructura del informe es como sigue: en el capítulo 2 se introducirán los conceptos importantes relacionados con el tema en particular, los conceptos matemáticos y algoritmos involucrados. En el capítulo 3 se describe con más detalle el problema que se intenta resolver en el ámbito del criptoanálisis, así como los requisitos que una solución tendría que tener. En el capítulo 4 se detalla el cómo fue implementada una solución para este problema, así como también se explica la arquitectura y la estructura de la solución discutiendo algunos problemas que se fueron encontrando durante el desarrollo, y por último se argumenta el por qué esta solución cumple los requisitos específicados. Debido a que esta solución necesita ser validada en la práctica, el capítulo 5 explica un caso de uso particular de la aplicación. Finalmente, en el capítulo 6 de Conclusiones se realiza un breve resumen del trabajo, un análisis tanto de la solución como de los resultados y se específica la dirección en la cual se podría continuar el desarrollo de éste.

A continuación se definirán con más detalles los términos importantes para la comprensión del funcionamiento del sistema desarrollado.

Capítulo 2

Antecedentes

En este capítulo, se explicarán los conceptos necesarios que permiten una mejor comprensión del problema y de la solución desarrollada. A partir de acá, se abusará del lenguaje y se ocupará el término "supuesto" indistintamente del término "problema computacional".

2.1. Conceptos principales

Se explicarán los conceptos matemáticos más importantes que son la base para el entendimiento de los algoritmos útiles en criptografía, por lo cual también se dará una breve explicación sobre la criptografía, el criptoanálisis y los supuestos computacionales.

2.1.1. Grupos Abelianos

Definición 1 Un grupo abeliano es un par $(G, *)$ donde G es un conjunto, y $*$ una operación binaria en G tal que cumple las siguientes propiedades:

(i) Asociatividad: $\forall a, b, c \in G \; a * (b * c) = (a * b) * c$

(ii) Identidad: $\exists e \in G$ tal que $\forall a \in G \; a * e = a = e * a$

(iii) Existencia de inversos: $\forall a \in G, \exists a' \in G$ tal que $a * a' = e = a' * a$

(iv) Conmutatividad: $\forall a, b \in G \; a * b = b * a$

Existe una noción más general de un grupo, el cual es definido abandonando la propiedad de la conmutatividad definida anteriormente. Sin embargo, en las aplicaciones

criptográficas comunes sólo se usan grupos que poseen la propiedad de la conmutatividad.

Es posible demostrar ([5]) que en un grupo G el elemento identidad es único, y que el inverso de cualquier elemento en G también es único.

Comúnmente son usadas dos notaciones para las operaciones de un grupo G:

- Notación multiplicativa ($\cdot$) en donde la operación entre dos elementos $a, b \in G$ se escribe $a \cdot b$ o simplemente ab. El elemento neutro o identidad se escribe como 1_G o simplemente 1 y el inverso de $a \in G$ se escribe a^{-1}.
- Notación aditiva ($+$) en donde la operación entre dos elementos $a, b \in G$ se escribe $a+b$. El elemento neutro o identidad se escribe como 0_G o simplemente 0 y el inverso de $a \in G$ se escribe $-a$.

Se usará comúnmente la notación multiplicativa, salvo para el caso de las curvas elípticas ([8]). A continuación se presentarán algunas de las propiedades básicas de los grupos:

Teorema 1 Sea $(G, \cdot)$ un grupo abeliano. Para todo $a, b, c \in G$ se tiene:

(i) Si $ab = ac$ entonces $b = c$

(ii) La ecuación $ax = b$ tiene una única solución $x \in G$

(iii) $(ab)^{-1} = a^{-1}b^{-1}$

(iv) $(a^{-1})^{-1} = a$

Se define para $a \in G$ y $n \in \mathbb{N}$,$a^n = 1 \cdot \underbrace{a \cdot a \cdot a ... \cdot a}_{n\ veces}$ y $a^{-n} = (a^{-1})^n$

Teorema 2 Sea $(G, \cdot)$ un grupo abeliano. Entonces para todo $a, b \in G$ y $k, l \in \mathbb{Z}$ se tiene:

(i) $(a^l)^k = a^{kl} = (a^k)^l$

(ii) $a^{k+l} = a^k a^l$

(iii) $(ab)^k = a^k b^k$

Se puede ver una idea de demostración de estos teoremas básicos en [5].

Un grupo abeliano puede ser trivial, lo que significa que sólo contiene la identidad.

Definición 2 Sea un grupo abeliano $(G, \cdot)$ finito. Se dice que $n \in \mathbb{N}$ es el orden de G si $n = |G|$, es decir, si n es la cardinalidad de G.

Definición 3 Sea $(G, \cdot)$ un grupo abeliano, y $H \subseteq G$. Entonces se dice que H es un subgrupo de G si para cada $a, b \in H$, $ab \in H$ y $a^{-1} \in H$. Dicho de otra forma H es un subgrupo de G si $H \subseteq G$ y H es un grupo. Es posible mostrar que ambas definiciones son equivalentes (ver por ejemplo [6]).

Definición 4 Sea G un grupo abeliano y $a \in G$. Se dice que $< a >= \{a^i, i \in \mathbb{Z}\}$ es el subgrupo de G generado por a. Para una demostración de por qué es un subgrupo, ver [5].

Definición 5 Sea G un grupo abeliano. Se dice que G es cíclico si existe un elemento $g \in G$ tal que $< g >= G$.

Teorema 3 Sea G un grupo abeliano finito y sea $a \in G$. Sea n el orden de G entonces $a^n = 1$. Ver demostración en [5].

A continuación se mostrarán dos grupos relevantes en el tema a desarrollar:

2.1.1.1. Grupo de los enteros módulo n bajo la multiplicación ($\mathbb{Z}_n^*$) y suma ($\mathbb{Z}_n$)

Definición 7 Divisibilidad. Sean $a, b \in \mathbb{Z}$. Se dice que $a|b$ si y sólo si $b = ak$ donde $k \in \mathbb{Z}$. En este caso se dice que a divide a b, que b es un múltiplo de a o que b es divisible por a.

Definición 8 Máximo común divisor (mcd o gcd en inglés). Sea $a, b \in \mathbb{Z}$. Se dice que $d = gcd(a, b)$ si y sólo si $d|a$, $d|b$ y si existe $g \in \mathbb{Z}$ tal que $g|a$ y $g|b$ entonces $g|d$.

Definición 9 Congruencia módulo. Sean $a, b \in \mathbb{Z}$ y $n \in \mathbb{N}$. Se dice que $a \equiv b \mod n$ si y sólo si $n|b - a$.

La relación $\equiv$ es una relación de equivalencia sobre $\mathbb{Z}$, la cual define un conjunto de clases de equivalencia, cuyos representantes están en $\mathbb{Z}_n = \{i, 0 \leq i < n\}$. Para más detalles ver [5].

Se puede dotar a $\mathbb{Z}_n$ con la operación aditiva de los enteros, formando un grupo abeliano cíclico de orden n (en [5] se encuentran la verificación de algunas de las propiedades que cumple $\mathbb{Z}_n$ como grupo aditivo). Por ejemplo, la asociatividad, conmutatividad, neutro e inverso son heredadas de $\mathbb{Z}$, y la cerradura de la operación está dada por la relación de congruencia (ej en $\mathbb{Z}_2$, $1 + 1 \equiv 2 \equiv 0 \mod 2$).

Es posible dotar además a $\mathbb{Z}_n$ con la operación multiplicativa de los enteros. Sin embargo, esto no asegura que la estructura resultante sea un grupo, pues si bien la mayoría

de las propiedades las hereda de $\mathbb{Z}$, no todos los elementos poseen un inverso multiplicativo. El siguiente teorema, demostrado en [5], permite definir un grupo con la operación multiplicativa de los enteros:

Teorema 4 Sea $a \in \mathbb{Z}_n$. La ecuación $ax \equiv 1 \mod n$ tiene solución si y sólo si $gcd(a,n) = 1$ y está únicamente determinada módulo n.

Con lo anterior se define $\mathbb{Z}_n^* = \{i; gcd(i,n) = 1\}$ y se dota a este conjunto con la operación multiplicativa de los enteros. Es fácil ver que $\mathbb{Z}_n^*$ es un grupo y que el orden de $\mathbb{Z}_n^*$ es distinto de n. Si $n = p$ primo, entonces el orden del grupo es $p-1$. Más generalmente se puede definir lo siguiente:

Definición 10 Se define la función phi de euler como: $\varphi(n) = |\mathbb{Z}_n^*|$. Dicho de otro modo, es la cantidad de enteros a menores que n que son relativamente primos ($gcd(a,n) = 1$) con n.

Teorema 5 Pequeño teorema de Fermat. Para todo primo p y para todo $a \in \mathbb{Z}_n$ se tiene que $a^p \equiv a \mod p$.

Una demostración se puede encontrar en [5] o [6]. Se puede demostrar usando el teorema 3, y la función phi de euler de manera muy sencilla.

2.1.1.2. Grupo de las curvas elípticas

Antes de poder definir las curvas elípticas con utilidades criptográficas, necesitamos definir el concepto de cuerpo:

Definición 11 Un cuerpo K es una estructura algebraica, que cumple que junto a una operación "aditiva" y "multiplicativa" (sin el neutro aditivo) es un grupo abeliano, y que la operación multiplicativa, tiene que ser distributiva con respecto a la adición, es decir, $\forall a, b, c \in K, a(b + c) = ab + ac$.

Por ejemplo, la estructura $\mathbb{Z}_p$ de los enteros módulo un primo es un cuerpo. Este cuerpo también es denotado $\mathbb{F}_p$.

Para dar una explicación geométrica sobre las curvas elípticas, primero se introducirán sobre el cuerpo de los reales.

Definición 12 Una curva elíptica E es el conjunto de soluciones de la ecuación:

$$E : Y^2 = X^3 + AX + B$$

junto con un punto O que representa el infinito, en donde las constantes $A, B \in \mathbb{R}$ deben satisfacer que:

$$4A^3 + 27B^2 \neq 0$$

A esta relación se le denomina Ecuación de Weierstrass.

Se define una operación $+$ sobre este conjunto. Sean P y Q dos puntos en una curva elíptica E. Sea L la recta que pasa por P y Q, y R otro punto que pasa por L y por E. Entonces $P + Q = R'$ en donde R' es la reflexión del punto R con respecto al eje x. Si se quiere sumar P con P, entonces L es la recta tangente a la curva, que pasa por P. Si se quiere sumar P con su reflexión, digamos P', entonces la recta intersecta en un punto O en el infinito. Este punto O se encuentra en todas las rectas verticales. Por esto, si sumamos P con O, se obtiene P, pues la recta que pasa por P y por O intersecta en P', y si obtenemos la reflexión de P' respecto al eje x, obtenemos P otra vez.

Con esto, es fácil intuir que E es un grupo abeliano con respecto a esta operación $+$. Una posible demostración de esto se puede encontrar en [8]. Fórmulas para encontrar la suma de dos puntos de manera mecánica se pueden encontrar en [8], pero no abordaremos sobre ellas.

Análogamente a cómo definimos una curva elíptica sobre $\mathbb{R}$, la podemos definir sobre $\mathbb{F}_p$, en donde las constantes A y B deben estar en $\mathbb{F}_p$, al igual que los puntos, y las operaciones son sobre $\mathbb{F}_p$. Es posible demostrar [8], que las curvas elípticas sobre $\mathbb{F}_p$ también son un grupo abeliano.

Sobre el tamaño que tienen las curvas elípticas hay un teorema mencionado en [8] que da cuenta de esto:

Teorema 6 Sea E una curva elíptica sobre $\mathbb{F}_p$. Entonces:

$$|E(\mathbb{F}_p)| = p + 1 - t_p$$

donde $|t_p| \leq 2\sqrt{p}$.

2.1.2. Criptografía

La criptografía es el área que estudia el desarrollo y el uso de técnicas matemáticas y computacionales que permiten la comunicación segura entre dos partes, con el objetivo de garantizar la privacidad, integridad, autenticidad y la no repudiación de la misma:

- Confidencialidad: Asegurar que ninguna persona pueda leer un mensaje, excepto el receptor original del mensaje. Por ejemplo, los mensajes secretos entre aliados en una guerra, de manera que los enemigos no pueden descifrar.
- Autenticidad: La capacidad de probar que un mensaje viene de una entidad autorizada. Por ejemplo, los bancos electrónicos tienen que verificar la autenticidad de las personas de los cheques a pagar/cobrar, de forma de que no sea posible falsificar la identidad de alguien.
- Integridad: Asegurar al recibidor del mensaje recibido que el mensaje no fue modificado con respecto al original.
- No repudiación: La capacidad de probar que el enviador del mensaje, realmente envió el mensaje.

Al hablar de criptografía debemos definir un esquema criptográfico. Un esquema criptográfico denota aquel/aquellos algoritmo/s que otorgue/n privacidad, integridad y/o autenticidad en una comunicación. Dado esto podemos definir, por ejemplo, un esquema de encriptación como un esquema criptográfico que busca conseguir privacidad, y un esquema de firmas, como aquel esquema criptográfico que busca autenticidad en la comunicación.

2.1.3. Supuestos computacionales

Los esquemas criptográficos se construyen en base a supuestos computacionales, problemas que se conjetura son difíciles de resolver, es decir, problemas para los cuales, actualmente, no existe un algoritmo para resolver el problema en un tiempo razonable. Por ejemplo, se conjetura que el problema de factorizar un número con dos factores primos, cada uno de trescientos dígitos, por ejemplo, es difícil, pues no existe algoritmo que encuentre dichos factores rápidamente en un tiempo menor a 1 año.

La idea de que los esquemas criptográficos se basen en supuestos computacionales difíciles, es que, en general, si un supuesto es difícil de resolver, entonces el esquema es difícil de quebrar. Dicho de otra forma, si fuese posible quebrar el esquema en tiempo razonable, sería posible resolver el supuesto computacional en tiempo razonable. Por ejemplo, RSA [33] está basado en el supuesto de que la factorización es difícil. El esquema de encriptación RSA es, a grandes rasgos, como sigue: sean p y q primos, $n = pq$ y $e, d \in$

$\mathbb{Z}^*_{\varphi(n)}$ tales que $ed \equiv 1 \mod \varphi(n)$. Decimos que (n, e) es la clave pública y d es la clave privada. Para encriptar un mensaje $m \in \mathbb{Z}_n$ se hace $c \equiv m^e \mod n$. Para desencriptar un mensaje $c \in \mathbb{Z}_n$ se calcula $c^d \equiv m^{ed} \equiv m^1 \equiv m \mod n$, lo cual es cierto dado que $ed \equiv 1 \mod \varphi(n)$. Ahora, si factorizar fuese fácil, entonces un adversario cualquiera podría calcular p y q, con esto podría calcular $\varphi(n)$, y finalmente con esto podría calcular la clave privada d, es decir, podria obtener un mensaje original desde cualquier texto cifrado.

Existen cuatro supuestos relevantes en el tema a desarrollar: Logaritmo Discreto, Factorización, Colisiones de Hash y Preimágenes de Hash. Se eligieron estos cuatro puesto que son problemas conocidos, y son una base para otros supuestos. No se agregaron más para así acotar el problema.

El problema del Logaritmo Discreto trabaja sobre grupos matemáticos genéricos finitos, los cuales cuentan con un elemento generador del grupo. La factorización trabaja sobre los números naturales. Encontrar colisiones en hash, trabaja sobre cadenas de carácteres. Sus definiciones son:

1. Logaritmo Discreto (Discrete Logarithm, DL): Si $\mathbb{G}$ es un grupo cíclico de orden n y g es un generador del grupo, el problema es dado h en el grupo, encontrar el entero $x \in \mathbb{Z}_n$ tal que $g^x = h$.
2. Factorización: Dado $n \in \mathbb{N}$ el problema es encontrar $p_1, p_2, ..., p_s \in \mathbb{N}$ y $\alpha_1, \alpha_2, ..., \alpha_s \in \mathbb{N}$ tales que $\prod_{i=1}^{s} p_i^{\alpha_i} = n$, en otras palabras, la factorización entera de n.
3. Colisiones de Hash: Sea $H : \{0,1\}* \rightarrow \{0,1\}^n$ una función de hash. El problema se describe así: encontrar $x, y \in \{0,1\}*$ tales que $H(x) = H(y)$ y $x \neq y$.
4. Preimágenes de Hash: Sea $H : \{0,1\}* \rightarrow \{0,1\}^n$ una función de hash e $y \in \{0,1\}^n$. El problema se describe así: encontrar $x \in \{0,1\}*$ tal que $H(x) = y$.

2.1.4. Criptoanálisis

Se denomina criptoanálisis al estudio de técnicas matemáticas y computacionales con el objetivo de quebrar un esquema criptográfico, es decir, encontrar un mensaje secreto, o suplantar la identidad de una persona, por ejemplo. En palabras simples, el criptoanálisis se traduce en ataques hacia esquemas criptográficos o supuestos usados por los esquemas. A modo de ejercicio, pensemos que Alice y Bob se quieren comunicar privadamente, pero cada uno se encuentra lejos del otro. Deciden entonces usar un esquema de encriptación, y de esta forma encriptan los mensajes para que no puedan ser leídos por el resto. Sin embargo, supongamos que Eve ha conseguido los textos cifrados de su conversación, y logra, de forma muy hábil, obtener los mensajes originales. Podemos decir entonces que Eve ha realizado un ataque al esquema de encriptación.

En el tópico del criptoanálisis, el objetivo de este trabajo de memoria es diseñar e implementar una interfaz sobre un sistema distribuido que permita seleccionar y usar varios algoritmos criptoanalíticos sobre problemas dados, es decir, un sistema que permita realizar ataques para vislumbrar la seguridad de ciertos esquemas criptográficos.

2.2. Arquitecturas Estándares

2.2.1. Amazon Web Services

Amazon Web Services (AWS) provee recursos y servicios computacionales que se pueden usar para armar aplicaciones en unos minutos, pagando sólo lo que se usa. Por ejemplo, se puede arrendar un servidor en AWS con el cual es posible conectarse a él, configurar, y correr de la misma forma con la que se haría con un servidor físico. La diferencia es que el servidor virtual corre sobre una red a escala mundial, manejado por AWS.

Respaldado por la red AWS, un servidor virtual puede hacer cosas que un servidor físico no podría, como por ejemplo, la capacidad de poder escalar automáticamente a múltiples servidores, cuando la demanda de la aplicación lo necesite.

Con los servidores virtuales de AWS se puede hacer casi todo lo que se puede hacer con hardware físico: páginas web, aplicaciones, bases de datos, aplicaciones móviles, campañas de email, análisis de datos distribuido, almacenamiento de fotos, videos, etc., y redes privadas. Se listarán a continuación las categorías de funcionalidad ofrecidas por AWS:

- Servicios de administración
  - Acceso e identidad
  - Monitoreo
  - Implantación y automatización
- Servicios de aplicación
  - Búsqueda
  - Workflow
  - Mensajería
  - Distribución de contenido
  - Transcodificación de medios
  - Computación distribuida
- Servicios base
  - Cómputo
  - Almacenamiento
  - Bases de datos
  - Redes

Para cada categoría, existen uno o más servicios. Información más detallada sobre todos los servicios se pueden encontrar en la documentación [46]. Para este trabajo se usa principalmente el servicio de cómputo y redes llamado Amazon Elastic Compute Cloud (Amazon EC2) que se describe a continuación.

2.2.1.1. Amazon EC2

Amazon Elastic Compute Cloud provee capacidad computacional de tamaño variable en la nube de AWS. Se puede usar EC2 para lanzar tantos servidores virtuales como sea necesario, configurar la seguridad, las redes y gestionar el almacenamiento. EC2 permite que la aplicación sea capaz de escalar para manejar cambios en los requerimientos o en la demanda de las personas.

Amazon EC2, a grandes rasgos, provee las siguientes características:

- Capacidad de crear ambientes computacionales virtuales, llamados comúnmente *instancias.*
- Plantillas pre-configuradas para las instancias, denominadas *Amazon Machine Images (AMIs)* que contienen la configuración para los servidores, incluyendo el sistema operativo y software adicional.
- Distintos tipos de configuraciones de CPU, memoria, almacenamiento y capacidad de red para las instancias, conocido como *tipos de instancias o instance types.*
- Información segura de login usando *key pairs*, donde AWS mantiene la clave pública, y el cliente la clave privada en un lugar seguro.
- Volúmenes de almacenamiento para datos temporales que son borrados en el momento de parar o terminar la instancia, conocidos como *instance store volumes*.
- Almacenamiento persistente para los datos usando Amazon Elastic Block Store (Amazon EBS), conocidos como *volumenes Amazon EBS.*
- Múltiples ubicaciones físicas para los recursos, tales como las instancias y los volúmenes EBS, conocidos como *regiones y Availability Zones.*
- Un firewall que permite específicar los protocolos, puertos, y rangos de IP de origen que pueden alcanzar las instancias, usando *security groups.*
- Direcciones IP estáticas para la computación dinámica en la nube, conocidos como *direcciones IP elásticas o Elastic IP Addresses.*
- Metadata, conocidas como *tags*, que se pueden crear y asignar a los recursos Amazon.

- Redes virtuales que se pueden crear y están lógicamente aisladas del resto de la nube en AWS y que se pueden conectar a otras redes, conocidas como *virtual private clouds (VPCs).*

En [45] se encuentran más detalles sobre Amazon EC2.

Los tipos de instancias disponibles pueden ser de 6 tipos:

- Instancias Micro.
- Instancias de propósito general.
- Instancias optimizadas para el cómputo.
- Instancias GPU.
- Instancias optimizadas para la memoria.
- Instancias optimizadas para almacenamiento.

Las instancias optimizadas para el cómputo están pensadas para aplicaciones que se beneficien de un alto poder computacional. Las instancias C3 son la última generación de estas instancias, en el cual cada CPU o CPU virtual (vCPU) es un hardware hyper-thread de un procesador Intel Xeon E5-2680v2 (Ivy Bridge) a 2.8 GHz. Estas instancias soportan una red avanzada que entrega mejores latencias entre instancias y un mejor rendimiento en la cantidad de paquetes por segundo. Por esto, son las instancias que se eligieron para armar el cluster. En [47] se puede ver con más detalle el enfoque del resto de las instancias.

Otro concepto importante en lo que respecta a la creación de un cluster de instancias, es el de *placement group.* Un placement group es un agrupamiento lógico de instancias dentro de un mismo Availability Zone. La funcionalidad de estos placement groups es la de permitir a las aplicaciones la obtención de una banda ancha de bisección completa y una red de baja latencia, requerida para la fuertemente acoplada comunicación nodo a nodo tipica de las aplicaciones de alto rendimiento. Para que una instancia pertenezca a un placement group, hay que crear la instancia en esta placement group.

Un placement group tiene ciertas limitaciones, por ejemplo, no puede tener instancias de distintos Availability Zones, y sólo ciertos tipos de instancias estan soportadas, entre ellas todas las Compute Optimized (optimizadas para el cómputo).

2.3. Herramientas útiles

En esta sección se describirá MPI, una herramienta útil en la implementación del sistema que permite otorgar y facilitar una comunicación distribuida necesaria para el sistema. Además, se presentará a Play!, el framework web con el cual se trabajará para desarrollar el sistema.

2.3.1. Message Passing Interface (MPI)

MPI es una específicación de una interfaz de paso de mensajes. Se dirige principalmente al modelo de programación por paso de mensajes, en donde los datos son movidos del espacio de direcciones de un proceso, al espacio de direcciones de otro proceso a través de operaciones cooperativas en cada proceso. MPI es una específicación, no una implementación; hay diversas implementaciones disponibles de MPI.

Las principales ventajas de establecer un estándar de paso de mensajes, son portabilidad y facilidad de uso. En un ambiente de comunicación con memoria distribuida, en donde las rutinas de alto nivel y/o las abstracciones están construidas sobre rutinas de bajo nivel de paso de mensajes, los beneficios de la estandarización son particularmente evidentes.

Las principales metas de MPI son:

- Permitir comunicacion eficiente: evitar copiado de memoria a memoria, permitir la superposición de cómputo y comunicación.
- Permitir implementaciones que puedan ser usadas en ambientes heterogéneos.
- Asumir una interfaz confiable de comunicación entre dos o más partes.

2.3.1.1. Conceptos principales

MPI define una gran variedad de funcionalidades. Los siguientes conceptos ayudan a comprender y dar un mejor contexto de los objetivos principales:

Comunicador

Los objetos comunicadores conectan un grupo de procesos en una sesión MPI. Cada comunicador da a cada proceso contenedor un identificador independiente y arregla sus proceso contenedores en una topología ordenada. MPI también tiene grupos explícitos, pero éstos son principalmente útiles para organizar y reorganizar grupos de procesos

antes de que otra comunicación sea hecha. MPI entiende operaciones de intercomunicación entre comunicadores unidireccionales y bilaterales.

Los comunicadores pueden ser particionados usando varios comandos MPI. Entre esos comandos se incluye MPI_COMM_SPLIT, en donde cada proceso se une a uno de varios sub-comunicadores coloreados, declarándose a sí mismo, que tiene ese color.

Comunicación punto a punto

Una cantidad importante de funciones de MPI, involucran la comunicación entre dos procesos específicos. Un ejemplo popular es MPI_Send, el cual permite que un proceso específico, mande un mensaje a un segundo proceso específicado. Las operaciones punto a punto son particularmente útiles en comunicaciones irregulares o con patrones específicos, como por ejemplo, en una arquitectura con paralelización de datos, en donde cada procesador intercambia rutinalmente regiones de datos con procesadores específicos entre pasos de cálculo, o en una arquitectura master-slave en donde el master envía datos nuevos de una tarea cada vez que la tarea anterior es completada.

Comunicación colectiva

Funciones colectivas involucran una comunicación entre todos los procesos en un grupo de procesos (lo cual puede significar todos los procesos, o un subconjunto definido por el programa). Una función típica es la llamada a MPI_Bcast (nombre corto para “broadcast”). Esta función toma datos de un nodo, y lo manda a todos los procesos en el grupo de procesos. Una operación inversa, es la llamada a MPI_Reduce, la cual toma datos de todos los procesos de un grupo, realiza una operación (como sumar, por ejemplo), y guarda el resultado en un nodo. Reduce es a menudo útil al comienzo o al fin de una larga computación distribuida, en donde cada procesador opera sobre una parte de los datos y luego los combina en un resultado.

Tipos de datos derivados

Muchas funciones de MPI requieren que se especifique el tipo dato a enviar entre procesos. Esto es así debido a que aquellas funciones proceden dependiendo del tipo especificado. Si el tipo de dato es estándar, como int, char, double, etc., se pueden usar tipos de datos predefinidos de MPI como MPI_INT, MPI_CHAR, MPI_DOUBLE. También es posible definir en MPI tipos de datos personalizados, por ejemplo, para una struct cualquiera en C.

2.3.2. Play! Framework

Play! es un framework web que está escrito tanto en Scala como en Java, por lo cual soporta ambos lenguajes para desarrollar las aplicaciones web pertinentes. Sigue el patrón MVC o modelo-vista-controlador. Es liviano y stateless, con una arquitectura orientada a la web.

Play! está orientado al desarrollador, pues el código fuente se compila automáticamente si ocurre un cambio en algún archivo mientras este corriendo el servidor, teniendo sólo que actualizar la página web para observar los cambios. Como está escrito sobre Scala, Play! es seguro en tipos (o type-safe) pues ayuda a prevenir errores de tipos.

Como Play! está construido sobre Akka [26], usa un modelo totalmente asíncrono. Si a esto se le suma el hecho de que Play! no preserva estados, se tiene que Play! escala de manera simple y predecible.

Para más detalles sobre Play!, usos y funcionalidades ver [27].

2.4. Algoritmos considerados

En esta sección se entrega una revisión de los algoritmos existentes que permiten resolver supuestos computacionales relevantes para el sistema, como factorizar y encontrar el logaritmo discreto. Cada algoritmo posee una descripción de cómo funciona, una forma de paralelizar este algoritmo de manera de poder implementarlo en un sistema distribuido y el tiempo secuencial que éste toma en función de la entrada, así como implementaciones secuenciales o paralelizadas del algoritmo.

Es importante destacar que si bien no todos estos algoritmos estarán presentes en el sistema desarrollado, esta sección sirve como una guía que muestra los algoritmos existentes junto con algunas implementaciones para que en un trabajo futuro éstos puedan ser puestos en la plataforma.

2.4.1. Factorización

A continuación se revisarán los algoritmos existentes para factorizar. Estos van desde el algoritmo más simple como lo es Trial Division, hasta los más sofisticados como Number Field Sieve.

2.4.1.1. Trial Division

Descripción

Este método para factorizar es prácticamente fuerza bruta, y es por ende el más ineficiente de los presentados para encontrar la factorización de un número, pero es el más fácil y simple de implementar.

Trial Division consiste básicamente en recorrer los números mayores que 1 y menores que n, siendo n el número a factorizar, de forma de encontrar sus factores. Sin embargo, es posible encontrar un factor de n sólo recorriendo los primeros $\sqrt{n}$ números, pues si $n = ab$, $a > \sqrt{n}$ y $b > \sqrt{n}$, entonces $ab > n$, lo cual no puede ser, y uno de sus factores tiene que ser menor a $\sqrt{n}$. Esto se puede mejorar un poco más recorriendo sólo los números menores a $\sqrt{n}$ tales que sean primos.

Paralelización

Paralelizar este algoritmo también es muy simple: si se tienen disponible p procesadores, entonces a cada uno se le asigna una cantidad de números a recorrer, de forma que cada procesador busque factores en rangos distintos.

Tiempo

En cuanto al tiempo, es fácil ver que hace $O(\sqrt{n})$ operaciones, lo cual es muy lento para cuando n sea, por ejemplo, del orden de 1000 bits (que es lo que suele usarse en aplicaciones criptográficas), pero es muy simple de implementar comparado con el resto, por ende este algoritmo se implementó en el sistema.

2.4.1.2. Pollard's rho

Descripción

Este método propuesto por John M. Pollard, está basado en el uso de una secuencia semi-aleatoria para encontrar los factores. La secuencia usada es generada por un polinomio f de coeficientes enteros, por ejemplo, $f(x) = x^2 + a$, donde a es una constante pequeña, y usualmente igual a 1. Si n es el número a factorizar, consideremos $x_1 \in \mathbb{Z}_n$ y la secuencia $x_2, x_3...$ en donde $x_{i+1} = f(x_i) \mod n$.

Buscamos dos valores distintos x_i, x_j con $i < j$ tales que $mcd(x_i - x_j, n) > 1$, es decir, un factor de n. Para esto, notemos que si p es un factor primo de n, y supongamos

que $x_i \equiv x_j \mod p$, entonces al aplicar f a la congruencia, tenemos que $f(x_i) \equiv f(x_j) \mod p$. Como $x_{i+1} = f(x_i) \mod n$ y $p|n$ se tiene que

$$x_{i+1} \mod p = (f(x_i) \mod n) \mod p = f(x_i) \mod p$$

Luego, $x_{i+1} \equiv x_{j+1} \mod p$. Más generalmente, $x_{i+k} \equiv x_{j+k} \mod p$ para todo entero $k > 0$ o si $l = j - i$, $j' > i' \geq i$ y $j' - i' \equiv 0 \mod l$ entonces $x_{i'} \equiv x_{j'} \mod p$. Finalmente notemos que tiene que existir un primer par x_i, x_j tal que $x_i \equiv x_j \mod p$, para el cual $x_1, x_2, x_3 ..., x_i$ son todos distintos módulo p, y luego de esto, se forma un ciclo módulo p de tamaño $j - l$ por lo destacado anteriormente. Para simplificar el algoritmo se busca i tal que $x_i \equiv x_{2i} \mod p$, el cual se puede demostrar que siempre existe.

Paralelización

Una forma de lograr computar paralelamente el algoritmo, es entregando un número x_0 distinto a cada procesador, de manera que trabajen independientemente entre ellos, y que todos paren si algúno de los procesos encuentra el factor, pero esta forma no entrega una ganancia lineal. En contraste, el método en [2] si posee una ganancia lineal.

Tiempo

En [33] se realiza un análisis aproximado del tiempo del algoritmo, el cual resulta tener una complexidad de $O(\sqrt{p})$ o $O(\sqrt[4]{n})$ pues $p < \sqrt{n}$. Este tiempo es sustancialmente mejor que el algoritmo de fuerza bruta o Trial Division, sin embargo no es el más rápido existente. Existen implementaciones en [40], [41] por ejemplo.

2.4.1.3. Pollard's $p - 1$

Descripción

Este algoritmo también fue propuesto por John M. Pollard. La idea es la siguiente: sea $n \in \mathbb{N}$ y p un factor de n, y sea B tal que cada potencia prima de $p - 1$ sea menor que B. Luego, es fácil ver que $p - 1|B!$. Sea $a = 2^{B!} \mod n$. Como $p|n$ se tiene que:

$$a \equiv 2^{B!} \mod n$$

Por el teorema de Fermat ([5]):

$$2^{p-1} \equiv 1 \mod p$$

Luego, como $p-1|B!$

$$a \equiv 1 \mod p$$

Sea $d = gcd(a-1, n)$. Si $d > 1$ y $d < n$, entonces podemos intentar factorizar d para encontrar un factor de n.

Paralelización

El uso de una segunda fase para este algoritmo genera mejoras considerables en tiempo y una posibilidad para paralelizar. Más detalles en [39].

Tiempo

Nuevamente en [33] se muestra un análisis del tiempo del algoritmo, el cual es $O(B \log B(\log n)^2 + (\log n)^3)$. Si B es $O((\log n)^i)$ con i fijo, entonces el algoritmo es polinomial (en $\log n$), pero con una probabilidad de éxito muy pequeña. Por el otro lado si B es $O(\sqrt{n})$ el algoritmo tiene éxito garantizado, sin embargo, no sería mejor que Trial Division, lo cual es un problema. Otro contra del algoritmo, es que requiere que $p-1$ posea primos pequeños, sin embargo, es fácil armar un módulo RSA $n = pq$ tal que $p-1$ y $q-1$ sean primos grandes. Implementación secuencial en [41].

2.4.1.4. Lenstra's Elliptic Curve Method

Descripción

Este método propuesto por Hendrik Lenstra, está basado en el algoritmo Pollard's p-1 de Pollard; es un análogo de éste pero en curvas elípticas (vea [8]). Si n es el número a factorizar, $P = (a, b)$ un punto en $\mathbb{Z}_n^2$ aleatorio y $A \in \mathbb{Z}_n$ aleatorio. Con esto tenemos $B = b^2 - a^2 - Aa \mod n$, con el cual tenemos una curva $E : Y^2 = X^3 + AX + B$. Si tenemos $Q = i!P$, es fácil calcular $(i+1)!P$ pues es igual a $(i+1)Q$. Sin embargo, como estamos trabajando con curvas elípticas sobre un grupo y no sobre un cuerpo, en el momento de hacer alguna operación sobre los puntos, tendremos que hacer divisiones (es decir, inversiones), lo cual no siempre está definido en $\mathbb{Z}_n$, y requiere la coprimalidad entre n y el número a ser invertido. Por esto es que pueden pasar 3 cosas:

- Es posible calcular $(i+1)Q$, con lo cual se continúa.
- El cálculo de $(i+1)Q$ falla, es decir, se intentó invertir un número k que no tenía inverso módulo n. Si esto pasa, significa que $d = gcd(k, n) \neq 1$. Si $d \neq n$ entonces encontramos un factor no trivial de n.
- El cálculo de $(i+1)Q$ falla, pero $d = n$. En este caso hay que probar con otra curva pues se llega a un punto muerto.

Paralelización

Una forma de paralelizar el proceso, es pasarle a cada procesador/proceso, una curva distinta, de manera que trabajen independientemente. Otra forma de paralelizar ECM se encuentra descrito en [39], el cual usa una segunda fase.

Tiempo

Según [8] ECM tiene un tiempo de ejecución promedio de $O(e^{\sqrt{2(\log p)(\log \log p)}})$, es decir, es un algoritmo de tiempo subexponencial. De hecho, es el tercer algoritmo más rápido para factorizar. Implementaciones existentes en [34] y [41].

2.4.1.5. Factorización por diferencia de cuadrados

La idea de los algoritmos más rápidos para factorizar (Quadratic Sieve y Number Field Sieve) se basa en el uso de la diferencia de cuadrados:

$$x^2 - y^2 = (x - y)(x + y)$$

Por ejemplo, si encontramos b tal que $kn + b^2 = a^2$ para algún k, podemos calcular $gcd(n, a - b)$ y conseguir un factor de n. Este problema es equivalente a encontrar a y b tales que n. Para hacer esto, seguimos tres pasos fundamentales:

1. Encontramos $a_1, a_2, ..., a_i$ tales que $c_i \equiv a_i^2 \mod n$ sea un producto de primos pequeños (menores a un cierto B).
2. Escogemos $c_{i_1}, c_{i_2}, ..., c_{i_s}$ de los c_i tales que todo primo que aparezca en su producto, tenga una potencia par. De esta forma $c_{i_1} c_{i_2} ... c_{i_s} = b^2$ es un cuadrado perfecto.
3. Sea $a = a_{i_1} a_{i_2} ... a_{i_s}$ y calculamos $d = gcd(d, a - b)$. Por construcción es fácil ver que $a^2 \equiv b^2 \mod n$, así que hay una probabilidad razonable de que d sea un factor no trivial de N.

El tercer paso es sólo un cálculo de máximo común divisor, el cuál toma tiempo $O(\ln n)$, el segundo paso es la resolución de un sistema de ecuaciones líneales sobre el anillo de los enteros módulo 2 (como se puede ver en [8]), así que el problema radica principalmente en el primer paso: encontrar los valores a_i. Quadratic Sieve y Number Field Sieve son algoritmos que realizan este primer paso.

2.4.1.6. Quadratic Sieve

Descripción

Este algoritmo, inventado por Carl Pomerance, se encarga de realizar el primer paso de una factorización por diferencia de cuadrados. La idea está en usar una criba e ir reduciendo los términos en ella. Sea F el polinomio definido por; $F(x) = x^2 - n$, donde n es el número a factorizar. Escogemos un valor de a inicial que sea un poco más grande que $\sqrt{n}$, es decir, $a = \lfloor \sqrt{n} \rfloor + 1$. Luego armamos la lista:

$$C = \{F(a), F(a+1), F(a+2), ..., F(b)\}$$

para un cierto entero b. Dada la forma del polinomio, cada elemento de esta lista es un residuo cuadrático. Ahora tenemos que asegurarnos de encontrar elementos en esta criba cuyos factores primos (o potencias primas) sean menores que un cierto B. Para ello, observemos que si p es un primo menor que B, entonces la ecuación de congruencias:

$$t^2 \equiv n \mod p$$

puede tener o no soluciones. Si no tiene soluciones, entonces p no divide a ningún elemento de la lista y lo descartamos. Si tiene soluciones, posee dos:

$$t = \alpha_p, t = \beta_p$$

(si $p = 2$ posee sólo una, $t = \alpha_p$). Luego, por construcción:

$$F(\alpha_p), F(\alpha_p + p), F(\alpha_p + 2p), ...$$

y

$$F(\beta_p + p), F(\beta_p + p), F(\beta_p + 2p), ...$$

son todos divisibles por p. Por ende, partiendo de α_p, cada p términos, hay un elemento en la lista C que podemos dividir por p. Análogamente, partiendo de β_p, cada p términos, hay un elemento en la lista C que se puede dividir por p. Finalmente, luego de recorrer todos los primos menores que B (y potencias), si hay elementos en la lista C que son iguales a 1, entonces quiere decir que inicialmente, ese elemento era un número cuyos factores primos, eran todos menores que B, y tomamos estos elementos como nuestros a_i.

Paralelización

Una forma de paralelizar esto, para introducirlo a un entorno distribuido, es tener una memoria compartida por los núcleos a usarse, que contenga la lista inicial C, y en la que cada núcleo procese una fracción de los primos menores que B. De esta forma, aceleramos el trabajo de la selección de los primos pequeños.

Tiempo

El tiempo que toma factorizar un número es, según [35], $O(e^{\sqrt{(\ln n)(\ln \ln n)}})$, es decir, levemente más rápido que ECM. Es el más rapido para factorizar números hasta del orden de 2^{350}. Implementacion distribuida (es decir, que se ejecuta en múltiples computadores que se comunican mediante una red) en [42] y paralela (es decir, que se ejecuta en varios o todos los núcleos, en un mismo computador) en [34].

2.4.1.7. Number Field Sieve

Descripción

Este algoritmo también se basa en la factorización usando diferencia de cuadrados. La idea es trabajar sobre un anillo "más grande" que el de los enteros. Para factorizar un número n, primero encontramos un entero m distinto de 0 y un polinomio mónico irreducible f de grado pequeño, que satisfaga:

$$f(m) \equiv 0 \mod n$$

Una primera aproximación para encontrar eficientemente este polinomio y este entero m, consta de elegir primero el grado d del polinomio, y elegir m entre $(n/2)^{1/d}$ y $n^{1/d}$. Luego, si escribimos n en base m tenemos que:

$$n = a_0 + a_1 m + ... + c_d m^d$$

La condición en m nos asegura que $c_d = 1$, así que podemos tomar $f(x) = a_0 + a_1 x + ... + x^d$.

Con esto, definimos el anillo $\mathbb{Z}[\beta] = \{c_0 + c_1\beta + c_2\beta^2 + ... + c_{d-1}\beta^{d-1} \in \mathbb{C} : c_0, c_1, ..., c_{d-1} \in \mathbb{Z}\}$ donde β es una raíz de f. Luego encontramos pares $(a_1, b_1), ..., (a_k, b_k)$ que satisfacen que $\prod_{i=1}^{k}(a_i - b_i m) = A^2$ sea un cuadrado en $\mathbb{Z}$ y $\prod_{i=1}^{k}(a_i - b_i\beta) = \alpha^2$ sea un cuadrado en $\mathbb{Z}[\beta]$ (para mayores detalles vea [35]). Por definición de $\mathbb{Z}[\beta]$:

$$\alpha = c_0 + c_1\beta + c_2\beta^2 + ... + c_{d-1}\beta^{d-1}$$

Como $f(m) \equiv 0 \mod n$, se tiene que (en el anillo $\mathbb{Z}[\beta]$):

$$m \equiv \beta \mod n$$

Así que, por un lado se tiene que (en el anillo $\mathbb{Z}[\beta]$):

$$A^2 \equiv \alpha^2 \mod n$$

y por otro lado, de la representación de α (en el anillo $\mathbb{Z}[\beta]$):

$$B = c_0 + c_1 m + c_2 m^2 + ... + c_{d-1} m^{d-1} \equiv \alpha \mod n$$

Y con esto $A^2 \equiv B^2 \mod n$.

Notemos que la elección del polinomio es importante pues de él depende gran parte de la eficiencia del algoritmo, así que otros métodos se desarrollaron para la selección del polinomio (vea [36]).

Tiempo

La complejidad del algoritmo es $O(e^{(\ln n)^{1/3}(\ln n)^{2/3}})$ (más detalles en [8]). Es el método más rápido para factorizar números de órdenes más grandes que 2^{450}. Existe una implementación en [43] con soporte para MPI, el cual permite la comunicación entre un cluster de procesadores mediante paso de mensajes, y en [34] una implementación paralela en los núcleos de un computador.

2.4.2. Logaritmo Discreto

Análogamente con la sección anterior, se presentarán los algoritmos existentes que permiten encontrar el logaritmo discreto de un elemento de un grupo G cualquiera.

2.4.2.1. Fuerza Bruta

Descripción

El algoritmo más simple de implementar pero más ineficiente en términos de tiempo.

La idea es simplemente tomar g e ir calculando $g^1, g^2, g^3, ...$, donde cada uno de estos términos es comparado con x para ver si $g^i = x$ y retornar i como el logaritmo discreto de x.

Paralelización

Para paralelizar esto, es posible que si tenemos varios cores, cada uno deba calcular desde g^{i_p} en adelante hasta tener una cierta cantidad de términos que depende de la cantidad de cores. i_p depende de cada procesador.

Tiempo

El tiempo de ejecución del algoritmo es claramente $O(n)$. A pesar de esta ineficiencia, se implementó este algoritmo el sistema, dada su simpleza de implementación.

2.4.2.2. Shanks' Algorithm

Descripción

Como su nombre lo dice, este algoritmo fue propuesto por Shanks [7], y es un trade-off entre tiempo y memoria.

Sea $m = \lceil \sqrt{n} \rceil$, y formamos dos listas:

$$L_1 = \{(k, g^{mk}) : k \in \{0, 1, ..., m-1\}\} \quad L_2 = \{(k, xg^{-k}) : k \in \{0, 1, ..., m-1\}\}$$

Sean $(j, y) \in L_1$ y $(i, y) \in L_2$, es decir, elementos que tengan en ambas listas la misma segunda coordenada. Para esto basta ordenar las dos listas con respecto a su segunda coordenada.

Notemos que, como $(j, y) \in L_1$ y $(i, y) \in L_2$, tenemos que:

$$g^{mj} = y = xg^{-i}$$

es decir:

$$g^{mj+i} = x$$

Luego, si t es tal que $x = g^t$ con $0 \leq t \leq n-1$, entonces de la relación anterior:

$$t = mj + i \mod n$$

Tiempo

No es difícil de implementar este algoritmo de forma que tome un tiempo $O(\sqrt{n})$ (véase [8]), lo cual es mucho mejor que el algoritmo de fuerza bruta. Existe una implementación en [44].

2.4.2.3. Pohlig-Hellman Algorithm

Descripción

Sea $n = \prod p_i^{c_i}$. Supongamos que tenemos un primo q tal que:

$$n \equiv 0 \mod q^c$$

y

$$n \not\equiv 0 \mod q^{c+1}$$

Si $a = \log_g x$, sea $y = a \mod q^c$. Como $0 \leq y \leq q^c - 1$, la representación de y en base q se puede expresar como:

$$y = \sum_{i=0}^{c-1} a_i q^i$$

donde $0 \leq a_i \leq q - 1$. Además, por el teorema de la división tenemos que:

$$a = y + sq^c$$

para algún entero s. Luego,

$$a = \sum_{i=0}^{c-1} a_i q^i + sq^c$$

Definamos $x_0 = x$ y $x_j = xg^{-(a_0 + a_1 q + \ldots + a_{j-1} q^{j-1})}$. Primero es fácil ver que $x_{j+1} = x_j g^{-a_j q^j}$, y segundo, es posible demostrar que:

$$x_j^{n/q^{j+1}} = g^{a_j n/q}$$

Con esta relación, el problema se reduce a calcular los factores a_i, es decir, encontrar el logaritmo discreto de $x_i^{n/q^{j+1}}$ sobre el grupo de orden q generado por $g^{n/q}$. Finalmente, el logaritmo discreto de x está dado por el teorema chino del resto [5]: Si tenemos $a \mod p_i^{c_i}$ para cada i, entonces tenemos $a \mod n$.

Paralelización

Cada procesador se encarga de calcular $a \mod p_i^{c_i}$ para ciertos i, en donde cada procesador se encarga de calcular aproximadamente la misma cantidad.

Tiempo

El orden de este algoritmo para cada primo q de n es de $O(c\sqrt{q})$ ignorando factores logarítmicos, según [8]. Sin embargo, esto no es tan eficiente si n no tiene factores primos pequeños. Hay varias implementaciones de esto en la web, por ejemplo en [38].

2.4.2.4. Pollard's rho

Descripción

Este método propuesto por John M. Pollard, está basado en el uso de una secuencia aleatoria para encontrar el logaritmo discreto. Sea G un grupo cíclico de orden n, $g \in G$ tal que g sea un generador de G, e $y \in G$ el elemento al cual le queremos obtener el logaritmo discreto, es decir $m \in \mathbb{Z}_n$ tal que $g^m = y$. La idea está en generar una secuencia "aleatoria" mediante una función previamente escogida, y encontrar una colisión. Sea $\{S_1, S_2, S_3\}$ una partición de G, donde cada elemento de la partición tiene aproximadamente la misma cantidad de elementos y sea f tal que:

$$f(x,a,b) = \begin{cases} (yx, a, b+1) & si\, x \in S_1 \\ (x^2, 2a, 2b) & si\, x \in S_2 \\ (gx, a+1, b) & si\, x \in S_3 \end{cases}$$

Cada una de las tripletas que se forman, tienen que cumplir que $x = g^a y^b$. Comenzamos con una tupla inicial teniendo estra propiedad, digamos $(1,0,0)$. Observemos que si (x,a,b) satisface la propiedad, entonces $f(x,a,b)$ también la satisface. Por ende definamos:

$$(x_i, a_i, b_i) = \begin{cases} (1,0,0) & si\, i = 0 \\ f(x_{i-1}, a_{i-1}, b_{i-1}) & si\, i \neq 0 \end{cases}$$

Comparamos la tupla (x_{2i}, a_{2i}, b_{2i}) con (x_i, a_i, b_i) hasta que encontremos un valor de $i \geq 1$ tal que $x_{2i} = x_i$. Cuando esto pase, tenemos que:

$$g^{a_{2i}} y^{b_{2i}} = g^{a_i} y^{b_i}$$

Si m es el logaritmo discreto de y entonces:

$$g^{a_{2i}+mb_{2i}} = g^{a_i+mb_i}$$

Como G es cíclico de orden n entonces

$$a_{2i} + mb_{2i} \equiv a_i + mb_i \mod n$$

Lo cual puede ser descrito como:

$$m(b_{2i} - b_i) \equiv a_i - a_{2i} \mod n$$

Si $gcd(b_{2i} - b_i, n) = 1$ entonces podemos encontrar m como:

$$m \equiv (a_i - a_{2i})(b_{2i} - b_i)^{-1} \mod n$$

Paralelización

Análogamente al caso de factorización, para paralelizar este algoritmo es posible entregar a cada procesador una tupla inicial (x_0) distinta, de forma de que el primer procesador que lo encuentra, lo retorna. Sin embargo, este método no provee una ganancia lineal en tiempo, con respecto a la cantidad de procesadores. Por esto, el método descrito en [2] o en el apartado de pollard's rho para funciones de hash, sí provee una ganancia lineal.

Tiempo

El tiempo asintótico, según [8], que toma este algoritmo es de $O(\sqrt{p})$ donde p es el mayor de los factores primos de n. El tiempo asintotico es uno de los mejores. Implementaciones distribuidas de éste para resolver este problema en curvas elípticas se pueden encontrar en [37].

2.4.2.5. Index Calculus

Descripción

Este método tiene un parecido considerable a muchos de los mejores algoritmos para factorizar, y sólo se puede usar para calcular logaritmos discretos del grupo $\mathbb{Z}_p^*$. La

idea es la siguiente: Sea L una base de primos, es decir, una lista de primos menores a un cierto B. Se tienen dos fases, una fase de precálculo donde se calculan los logaritmos discretos de los primos de la base de primos, y una fase donde se calcula el logaritmo discreto de x.

Sea C un entero un poco más grande que B, digamos $C = B + 10$. Construimos C congruencias módulo p de la siguiente forma:

$$g^{y_j} \equiv p_1^{a_{ij}} p_2^{a_{2j}} p_3^{a_{3j}} ... p_B^{a_{Bj}} \mod p$$

Con $1 \leq j \leq C$. O equivalentemente:

$$y_j \equiv a_{1j} \log_g p_1 + a_{2j} \log_g p_2 + ... + a_{Bj} \log_g p_B \mod p - 1$$

Ahora ¿cómo escoger y_j?. Una forma fácil es escoger C elementos al azar, e intentar factorizar cada g^{y_j} sobre la base de primos. Ahora con estas C congruencias, esperamos encontrar una única solución que las satisfaga. Con esto, tenemos hecha la fase de precálculo al poseer los logaritmos discretos de la base de primos. Para la segunda fase, nos basamos en un algoritmo tipo Las Vegas. Sea s al azar tal que $1 \leq s \leq p-2$ y sea $z \equiv xg^s \mod p$. Intentamos factorizar z sobre la base de primos, y si es posible entonces podemos escribir la congruencia anterior como:

$$xg^s \equiv p_1^{a_1} ... p_B^{a_B} \mod p$$

O equivalentemente

$$\log_g x + s \equiv a_1 \log_g p_1 + ... + a_B \log_g p_2 \mod p - 1$$

Con esto, dado que tenemos los logaritmos de la base de primos, la factorización de z, y s, podemos reemplazar los valores conocidos, para despejar el logaritmo discreto de x.

Paralelización

La fase de precálculo se puede paralelizar fácilmente haciendo que cada procesador sólo obtenga un subconjunto de y_j e intente factorizar aquel subconjunto, en donde al final se obtienen todas las C congruencias calculadas por todos los procesadores.

Tiempo

El tiempo esperado que toma el algoritmo en la fase de precálculo es, en notación asintótica, de $O(e^{(1+o(1))\ \sqrt{(\ln p)(\ln \ln p)}})$, y el tiempo que tarda en la segunda fase es de $O(e^{(1/2+o(1))\sqrt{(\ln p)(\ln \ln p)}})$ según [8]. Como se puede ver, estos tiempos son menores que los de otros algoritmos, sin embargo no se han encontrado implementaciones de éste.

2.4.3. Colisiones de Hash

2.4.3.1. Fuerza bruta

Descripción

Es el método intuitivo para encontrar una colisión. Sea $f : \{0,1\}^* \to S$ una función, donde $|S| = n$. Para encontrar una colisión, basta probar con distintas entradas para f hasta encontrarla. Para esto, es necesaria una tabla de hash para ir guardando y comprobando estos $f(x)$.

Paralelización

Cada proceso o core, puede realizar el cálculo de la función de hash, y guardar estos valores en una memoria compartida o distribuida, para que otros procesos puedan verificar colisiones. Con esto se logra una ganancia lineal en tiempo.

Tiempo

Según [3], la cantidad esperada de entradas a probar es de $\sqrt{\pi n/2}$. Por ende, y dado que se deben guardar estos valores en una tabla de hash, se tiene una complejidad de $O(\sqrt{n})$ en tiempo, y de $O(\sqrt{n})$ en espacio, lo cual puede ser un espacio demasiado grande, para un n grande, incluso con una paralelización adecuada.

2.4.3.2. Pollard's rho

En este apartado, se explicarán métodos generales para encontrar colisiones sobre algoritmos de hash, ya que no dependen de la estructura interna de las funciones. Además, este enfoque de encontrar cualquier colisión, puede no ser útil en la realidad, sin embargo, en [2] se usa este enfoque para encontrar una colisiones útiles para falsificar firmas digitales.

Descripción

Este método toma un camino semi-aleatorio sobre algún conjunto finito S. Es muy parecido a los métodos que factorizan y calculan logaritmo discreto que llevan el mismo nombre. Sea $f : S \to S$ con $|S| = n$, $x_0 \in S$ y la secuencia $x_i = f(x_{i-1})$. Como S es finito, la secuencia anterior, eventualmente alcanzará un ciclo. Si x_l es el último punto antes de que comience el ciclo, entonces x_{l+1} está en el ciclo. Sea x_c el punto en el ciclo que precede inmediatamente a x_{l+1}. Así tenemos la colisión deseada pues $f(x_l) = f(x_c)$ y $x_l \neq x_c$.

Para detectar el ciclo, es posible usar el algoritmo de Floyd para esto [4], u otros métodos similares.

Paralelización

Para lograr una ganancia lineal en el tiempo secuencial del algoritmo, es necesario el uso de puntos distinguidos. Un punto distinguido, es uno que tiene una propiedad fácil de verificar, como la cantidad de ceros al principio de la cadena. Cada proceso parte con un x_0 distinto, y comienzan a calcular la secuencia anterior, hasta encontrar un punto distinguido. Este punto distinguido se guarda en una tabla de hash en una memoria compartida por los procesos, acompañado por el punto inicial y el tamaño del camino recorrido, y luego se reinicia el x_0 inicial junto con la secuencia. Cuando en la tabla de hash, se encuentra un punto distinguido ya guardado, entonces es muy probable que se haya encontrado una colisión. Para obtenerla, basta tomar el punto inicial con el camino más largo, aplicarle f hasta que el camino restante sea igual al punto con el camino más corto, y luego ir calculando f de los dos puntos en paralelo. Los puntos anteriores al distinguido, son los puntos que logran una colisión.

Tiempo

El tiempo secuencial es $O(\sqrt{n})$, pues recibe el mismo análisis que el tiempo del algoritmo anterior, sin embargo la complejidad en espacio puede llegar a ser constante, dependiendo del algoritmo para encontrar ciclos. El tiempo paralelo del algoritmo es $O(\sqrt{n}/p)$ donde p es la cantidad de núcleos, es decir, tiene una ganancia lineal con respecto a la cantidad de núcleos, pero la complejidad en espacio depende fuertemente de la cantidad de puntos distinguidos, pues es orden $O(\sqrt{k})$ si k es la cantidad de puntos distinguidos.

Este algoritmo fue implementado en el sistema desarrollado.

2.4.4. Preimágenes de Hash

Se describirá sólo el algoritmo de fuerza bruta genérico para cualquier función de hash. Esto debido a la simplicidad y la baja complejidad en espacio que este posee.

2.4.4.1. Fuerza bruta

Descripción

El método intuitivo de encontrar una preimagen. Sea $f : \{0,1\}^* \to S$ una función, donde $|S| = n$. Para encontrar una preimagen basta probar con entradas aleatorias para f hasta que $f(e) = y$, donde y es la imágen y e es la preimagen a encontrar.

Paralelización

Cada proceso o core, puede ir calculando entradas aleatorias independiente del resto, con lo cual se puede encontrar una preimagen más rápido.

Tiempo

Asumiendo que las funciones poseen una distribución aleatoria, la probabilidad de que una entrada a la función, de lugar a la imagen buscada es $1/n$, por ende el tiempo esperado para encontrar una preimagen es $O(n)$.

2.5. Soluciones existentes que realizan criptoanálisis

Con el motivo de verificar la existencia de algún sistema/aplicación/hardware que pueda resolver el problema que se describirá posteriormente, se ha realizado un estudio de aquellos sistemas que realicen criptoanálisis.

2.5.1. Máquinas personalizadas (Custom hardware attack)

Estas máquinas están específicamente diseñadas con circuitos integrados de aplicación específica (application-specific integrated circuits o ASIC) para realizar criptoanálisis.

Para montar un ataque de fuerza bruta criptográfico, se necesita una larga cantidad de computaciones similares: típicamente probando con una llave, revisando, por ejemplo, si la desencriptación da un mensaje con significado, y probar la siguiente llave si esto último no pasa. Los computadores pueden realizar estas operaciones a una velocidad de millones por segundo, y miles de computadores pueden ser configurados juntos en una red distribuida de computadores. Sin embargo, el número de operaciones crece exponencialmente con el tamaño de la llave y para muchos problemas, los computadores estándar no son suficientes. Por otro lado, muchos algoritmos criptográficos permiten

implementaciones rápidas en hardware, es decir, usando redes de circuitos lógicos. Los circuitos integrados son construidos basados en estas redes, y a menudo ejecutan algoritmos criptográficos cientos de veces más rápido, que en un computador de propósito general.

Una alternativa para los circuitos integrados, es el uso de FPGAs (field-programmable gate arrays), las cuales son más lentas pero pueden ser reprogramadas para diferentes problemas. COPACOBANA [9] usa estos circuitos.

Algunos ejemplos de máquinas personalizadas son:

EFF DES Cracker [11]

Es una máquina construida por EFF (Electronic Frontier Foundation) en 1998 para realizar una busqueda de fuerza bruta, en el espacio de llaves de DES [10]. El propósito de esto era demostrar que DES no era seguro. También es llamado Deep Crack.

Deep Crack fue construido en respuesta a “DES Challenge II-2” [12], con un costo de US$250.000 aproximadamente. Deep Crack desencriptó un mensaje encriptado de DES después de 56 horas de procesamiento, lo que demostró que atacar DES era una proposición muy práctica. Seis meses después, en respuesta a “DES Challenge III”, y en colaboración con distributed.net [13], desencriptaron otro mensaje encriptado de DES en menos de un día.

COPACOBANA [9]

El prototipo original de COPACOBANA (Cost-Optimized Parallel Code Breaker), consistia de 120 nodos FPGA, los cuales estaban conectados por un bus compartido, entregando un ancho de banda de 1.6 Gbps. COPACOBANA fue producido con un costo menor a 10.000 euros. Fue principalmente diseñado para aplicaciones y ataques criptoanalíticos simples, con un alto costo computacional, pero con requerimientos mínimos en comunicaciones y memoria local. COPACOBANA recupera, por ejemplo, una llave de DES en menos de 6.4 días en promedio.

En 2008, SciEngines, una compañia formada por los desarrolladores de COPACOBANA, introducen a RIVYERA S3-5000 [14], el sucesor directo de COPACOBANA con una nueva arquitectura, que permite calcular una llave de DES en menos de un día. RIVYERA posee 128 FPGAs.

Si bien ya existen máquinas personalizadas que realizan criptoanálisis de diversos esquemas, ninguna de estas máquinas poseen una interfaz que permita ejecutar de forma sencilla para un usuario casi cualquiera, un ataque criptoanalítico, ni menos de implementar fácilmente estos ataques en base a otros. Además, no era posible obtener o armar alguna máquina personalizada, debido a los recursos y al tiempo disponible para este trabajo.

2.5.2. Volunteer Computing

Volunteer Computing (o computación voluntaria) es un tipo de computación distribuida en donde los dueños de computadores donan sus recursos computacionales a uno o más “proyectos”.

El primer proyecto de computación voluntaria fue el GIMPS (Great Internet Mersenne Prime Search) [15], que comenzó en 1996. Fue seguido por distributed.net [13] en 1997.

Básicamente los sistemas de computación voluntaria tienen la siguiente estructura: un programa cliente se ejecuta en el computador del voluntario. Este contacta periódicamente con servidores dedicados al proyecto por internet, pidiendo tareas y reportando el resultado de tareas completadas. El sistema guarda el “crédito” de cada usuario: un valor numérico que mide cuánto ha hecho el computador del usuario para el proyecto.

Existen diversos problemas en los computadores de los voluntarios, que estos sistemas deben resolver: la heterogeneidad, cuando llegan y se van los usuarios, su disponibilidad intermitente y la necesidad de no interferir con rendimiento en un uso regular. Otros problemas relacionados con la correctitud de los resultados también se tienen que resolver: los computadores voluntarios pueden malfuncionar y entregar valores incorrectos, y los voluntarios pueden retornar intencionalmente valores incorrectos. Una solución para estos problemas, es realizar una computación replicada, en donde cada tarea es realizada en al menos dos computadores.

Algunos ejemplos de proyectos de computación voluntaria son:

BOINC [16]

BOINC (Berkeley Open Infrastructure for Network Computing) no es precisamente un proyecto de computación voluntaria, sino un sistema que entrega una infraestructura distribuida, independiente del procesamiento científico, de forma de proveer una plataforma para el desarrollo de proyectos de computación voluntaria. BOINC es mencionado porque es el sistema más usado.

distributed.net [13]

distributed.net es un esfuerzo mundial de computación distribuida que intenta resolver problemas de gran escala usando, ya sea CPUs inactivas o GPUs. En 1998 resolvió el “DES Challenge II-1” [12] en 41 días.

Actualmente está intentando resolver RC5-72 [17].

NFS@Home [18]

NFS@Home es un proyecto de investigación que usa computadores conectados a internet para realizar uno de los pasos del algoritmo de factorización, Number Field Sieve.

Al igual que con las máquinas personalizadas, un usuario no puede ejecutar un algoritmo específico como un ataque criptoanalítico, y si se desarrollase una interfaz para este tipo de sistemas junto con los algoritmos necesarios, se tendrian que conseguir suficientes usuarios que apoyen un proyecto de esta magnitud para lograr un poder de cómputo significativo lo cual es complejo.

2.5.3. Otros

A Java Framework for Cryptanalysis using GA (JFCGA) [19]

En este trabajo se realiza la implementación de un framework en java, que realiza criptoanálisis usando algoritmos genéticos e inteligencia artificial. Según el autor, este framework hace fácil el trabajo de migrar una aplicación Java, para que use inteligencia artificial en el criptoanálisis.

Si bien este framework implementa ciertos algoritmos para realizar criptoanálisis en cifradores de bloques, el propósito de este trabajo, no es la extensión, la escalabilidad ni la facilidad de uso de estos, sino la extensión de otros algoritmos para que puedan usar técnicas de los algoritmos genéricos y la inteligencia artificial de forma relativamente simple.

Quebra-Pedra [20]

Este framework java es una herramienta criptoanalítica, que permite realizar ataques de fuerza bruta y diccionario, con el objetivo de encontrar alguna contraseña con objetivos forenses. El autor aclama que esta herramienta es portable, extensible y eficiente, de forma que realizar un ataque criptoanalítico a un algoritmo especifico, no requiera de un mayor trabajo.

Sin embargo, si bien es extensible con aquellos ataques criptoanalíticos que intentan encontrar una “contraseña” de un algoritmo criptográfico especifico, ataques a otros supuestos o algoritmos criptográficos, por ejemplo factorización o RSA, no son posibles porque no están soportados por el sistema. Por otro lado, habría que crear una interfaz para que un usuario específico, pueda usar el framework sin necesariamente usar su propio computador.

Wisecracker [21]

Wisecracker es un framework enfocado a que un investigador pueda abstraerse en forma transparente del uso de los procesadores de diversas GPU, y/o de diversas CPU. Es importante destacar que con Wisecracker es posible usar computadores heterogéneos. Para el uso de los procesadores de la GPU o de la CPU, Wisecracker usa OpenCL [22], tanto de procesadores gráficos como de CPU. Para la operabilidad con distintos computadores, se usa MPI [23]. En resumen, Wisecracker es un framework que permite la abstracción, tanto de MPI como de OpenCL, y de esta forma un investigador se pueda enfocar en la implementación de los algoritmos criptoanalíticos.

Wisecracker no implementa algoritmos criptoanalíticos, ni una interfaz para el usuario, ni una plataforma escalable que permita el desarrollo de algoritmos criptoanalíticos que pueden depender de otros ya implementados. Sin embargo, Wisecracker podría haber sido usado dado que abstrae el uso de MPI, pero no fue usado ya que fue encontrado en las etapas finales del desarrollo de este trabajo.

Capítulo 3

Especificación del Problema

3.1. Descripción del problema a resolver

Los investigadores dedicados a la seguridad computacional ocasionalmente deben medir el tiempo que tarda en completarse cierto ataque criptoanalítico a un algoritmo (o esquema) criptográfico pues, en la práctica, más que el análisis asintótico teórico, lo que importa es el tiempo real que toman estos algoritmos. Este tiempo depende de la implementación específica que fue realizada, si hay optimizaciones de por medio, etc., y también depende de la arquitectura en la cual se ejecuta, pues por ejemplo, es distinto el tiempo que se tarda el algoritmo en un Intel Pentium II, un Intel Core i7 o un sistema distribuido, como un cluster de procesadores, homogéneo o heterogéneo, o un sistema de computación voluntaria. Es por esto que un ataque que en la teoría puede parecer completamente inviable (como un ataque de fuerza bruta que en teoría demoraría 100 años en terminar), en la práctica puede no ser así y terminar en un par de semanas, días, horas, minutos o incluso segundos.

Sin embargo, los investigadores no siempre tienen a disposición el hardware necesario para ejecutar algoritmos criptoanalíticos a gran escala, sino que generalmente trabajan directamente sobre sus computadores de escritorio, en los cuales pueden realizar ataques sobre parámetros de seguridad de pocos bits (por ejemplo, 32 bits), lo cual usualmente no aporta significado al estudio y/o medición que se encuentran realizando. Por otro lado, suponiendo que el investigador ya posee un hardware ad-hoc a sus necesidades (por ejemplo, un cluster de procesadores), es necesario tener además un software que permita realizar criptoanálisis. Por ejemplo, supongamos que el investigador tiene un cluster, y quiere implementar un ataque a RSA [33]. La forma directa es tener un algoritmo que encuentre la factorización del módulo RSA, y encuentre la clave privada teniendo esta factorización. Pero acá se tienen dos opciones: se implementa desde cero un algoritmo que factorice un número en sus factores, o se usa un algoritmo de factorización de código libre o de algún autor conocido. En general, no es recomendable “reinventar la rueda” por lo que se escoge un algoritmo implementado anteriormente (a no ser que se necesite una implementación muy específica, u optimizaciones que no se encuentran implementadas). El siguiente paso es adaptar este algoritmo de factorización (por ejemplo Number

Field Sieve) al cluster de procesadores. No obstante, este proceso de adaptación puede ser muy complejo y puede tomar un tiempo considerable, dado que hay que adaptar el algoritmo a las limitaciones de la arquitectura de destino, y puede ser aún más problemático si la arquitectura de destino es un sistema distribuido y el algoritmo encontrado es secuencial.

A esto se suma que realizar dicho proceso no es necesariamente flexible ni reusable para el resto de los investigadores. Por ejemplo, si hay un investigador que quiere usar un algoritmo para factorizar adaptado por otro investigador del campo, el primero debe conocer de antemano que el segundo investigador hizo una adaptación de este algoritmo, y además el primer investigador debe pedirle al segundo investigador acceso al código o a la libreria respectiva. Por otro lado, el código del segundo investigador puede no cubrir todas las necesidades del primer investigador, con lo cual éste tendría que cambiar el código antes de poder usarlo.

3.2. Relevancia de contar con una solución

Con una solución, un investigador dedicado a la seguridad podrá focalizarse en los algoritmos, y no en la implementación. Por ejemplo si el investigador realiza un estudio sobre el tiempo que tardan ciertos ataques criptoanalíticos basados en factorización, éste seguramente querrá usar algún algoritmo ya implementado, y ojalá este algoritmo se pueda usar casi inmediatamente, de forma de que las mediciones tomen menos tiempo. Por otro lado, si esto fuese reusable, muchos investigadores podrían usar estas implementaciones ya adaptadas, sin tener que reimplementarlas. En pocas palabras, los investigadores ahorran tiempo y pueden medir el tiempo sobre una arquitectura dada.

Otro objetivo que se puede cumplir al tener una solución, es que si existe abstracción del hardware y del software, estudiantes podrán apreciar aplicaciones prácticas de la criptografía, sin tener que pasar por la burocracia de adaptar estos algoritmos a sus propios computadores personales.

3.3. Requisitos de la solución

A continuación serán listados los requisitos:

- Cualquier usuario autorizado podrá ejecutar un ataque criptoanalítico de los algoritmos disponibles, sin inhabilitar con ello su propio computador. De esta forma, se intenta proveer una abstracción del hardware y del software para que los usuarios no tengan que pasar por la problemática tarea de encontrar un hardware apto, e implementar o adaptar un algoritmo criptoanalítico que resuelve algún supuesto computacional o quiebra algún esquema criptográfico. Otro problema que este requisito soluciona es que da reusabilidad de los algoritmos ya implementados, pues cualquier usuario podría usar algún algoritmo disponible que ya fue adaptado con anterioridad.

- Cualquier usuario autorizado podrá implementar un nuevo ataque criptoanalítico, que use como subrutina los algoritmos ya disponibles. Una variedad de esquemas criptográficos pueden ser atacados usando ataques criptoanalíticos a los supuestos computacionales de los cuales están formados. Por ejemplo si queremos atacar RSA y encontrar la clave privada, podemos buscar la factorización del módulo RSA, y con esto encontrar la clave privada de forma muy fácil. Este puede no ser el mejor tipo de ataque y pueden existir otros que no dependan de la resolución del supuesto computacional, sin embargo, dado que estos supuestos están resueltos de alguna forma, como una caja negra, resulta un ataque bastante simple de implementar.

- La solución tiene que ser de fácil acceso, y con un uso intuitivo. El fácil acceso para que se pueda acceder al sistema con relativa facilidad y con poca burocracia, de forma de haber una ganancia total en tiempo. El uso intuitivo es para hacer la labor mucho más fácil y que no sea una forma más difícil que implementarlo desde cero.

- La solución debiese ser extensible para que se puedan agregar otros algoritmos criptoanalíticos más sofisticados (por ejemplo, algunos de los descritos en el capítulo anterior) de forma fácil, ya sea mediante la adaptación de uno nuevo o mediante el uso de otros ya existentes. De esta forma la introducción de un nuevo algoritmo no será más difícil que adaptar un algoritmo nuevo (o implementar fácilmente uno en base a otros).

Capítulo 4

Descripción de la Solución

En este capítulo se describirá con detalle la solución implementada, cuyo código se encuentra disponible en https://bitbucket.org/efrias/cryptfront. En lo que sigue, existen dos partes fundamentales de la solución que hay que considerar para cada punto: un servidor web que atiende a los clientes y un cluster o sistema distribuido que realiza el cómputo de las tareas de los clientes.

4.1. Arquitectura del hardware

4.1.1. Servidor Frontend

El servidor web se encuentra sobre una instancia de propósito general que posee 1 vCPU o virtual CPU, con un poder de cómputo de 1 ECU (el cual equivale a 1.0-1.2 GHz de un procesador Xeon del 2007), 1.7 GiB de memoria, con un sistema operativo Amazon Linux 2013.09.2 y un almacenamiento persistente EBS de 8 GB. Se escogieron estas especíificaciones técnicas, pues la cantidad de usuarios que se tendría es pequeña y el procesamiento requerido por la aplicación no es intensivo.

4.1.2. Cluster AWS

El nodo principal del cluster es una instancia c3.large que posee 2 vCPUs, con un poder de cómputo de 7 ECU, 3.75 GiB de memoria RAM, con un sistema operativo Centos 6.4 x86_64 y un almacenamiento persistente EBS de 15 GiB, debido al poco espacio necesario para esta aplicación; se requiere más poder computacional que almacenamiento o RAM. También se tiene un disco de 15 GiB montado como un Network Filesystem (NFS), para tener un método centralizado de comunicarse con el resto de los nodos y el sistema web, y para tener los recursos necesarios en un sólo lugar, sin tener que duplicarlos en todos los nodos. Esta instancia puede ser fácilmente cambiada a una instancia tipo C3

con una mayor cantidad de vCPUs y mayor poder computacional, gracias a la facilidad que posee AWS para lanzar instancias con AMIs incluso personalizadas.

Con respecto al resto de los nodos, estos tienen que ser instancias Compute-Optimized para ponerlos dentro del mismo placement group y gozar de una comunicación con baja latencia. Nuevamente, gracias a la facilidad de AWS para escalar los sistemas y crear instancias, las instancias de los nodos pueden crearse, pararse o terminar, a medida que se necesita o se desea.

4.2. Arquitectura del software

4.2.1. Servidor Frontend

Este servidor estará encargado de dos cosas: funcionar como un servidor Web que sea la interfaz para los clientes, con una conexión a una base de datos MySQL montada en la misma instancia, y comunicarse con el cluster cuando el cliente requiera algún cálculo, como la resolución de un logaritmo discreto, la factorización de algún número o algo más complejo.

Para la interfaz web, se usa para su implementación un framework MVC (Modelo-Vista-Controlador) ya que facilita el trabajo y ya se es familiar con este tipo de framework. Para la conexión entre el servidor y el cluster, es necesario identificarse en el cluster con ssh de manera automática, y así hacer transparente a un usuario el hecho de que se está conectando con un cluster. Luego de identificarse, se le entregan al cluster los parámetros y el problema a resolver que el usuario ingresa en la plataforma web, y de esta forma iniciar un nuevo trabajo en el cluster. El servidor frontend entonces, espera una respuesta del cluster hasta que resuelva el problema o hasta que el usuario decida cancelar el trabajo. En tal caso, se debe conectar nuevamente con el cluster y además liberar los recursos tomados por la tarea cancelada.

4.2.2. Cluster AWS

Por el lado del cluster, una vez que el servidor frontend se conecta con el nodo principal del cluster, el servidor frontend encola el trabajo y le entrega los parámetros necesarios junto con la tarea a resolver cuando se encuentren recursos disponibles en el cluster, y el cluster manda el resultado al servidor cuando termine. También el servidor frontend se conecta con el nodo principal cuando se quiera cancelar una tarea, y para esto manda un *kill* al algoritmo ejecutándose en el cluster.

Es importante notar que en el cluster se tienen que encontrar implementaciones de los algoritmos criptoanalíticos que usa el sistema.

4.3. Diseño de la base de datos

La base de datos usada para el sistema es relativamente simple, pero necesaria, usando el sistema de gestion de base de datos MySQL. Esta compuesta por sólo dos tablas: una tabla **usuarios** y una tabla **estados**.

4.3.1. Tabla Usuarios

La tabla usuarios, a grandes rasgos, guarda los usuarios del sistema. Posee los siguientes campos:

id Tipo: int(10). Llave primaria de la tabla para numerar los usuarios.

nombre_usuario Tipo: varchar(255). Indica el nombre del usuario en el sistema.

password Tipo: varchar(64). BCrypt del texto plano de la contraseña del usuario.

semilla Tipo: varchar(128). Salt del BCrypt de la contraseña del usuario.

es_admin Tipo: tinyint(1). Indica si el usuario es administrador del sistema o no.

Para la plataforma era necesario contar con usuarios registrados para que el acceso al sistema sea restringido, debido a que pudiesen existir usuarios malintencionados que podrían darle un mal uso a este. Por esto, una persona que viene recién conociendo o entrando en el sistema, no puede acceder a él, a menos que se contacte con un administrador que cree el usuario. De esta forma, sólo entrarán al sistema personas que hayan sido validadas previamente con un encargado, y así reducir el mal uso que se le pueda dar a éste.

4.3.2. Tabla Estados

La tabla estados sirve para que los usuarios conozcan el estado de las tareas que han ejecutado. Posee los siguientes campos:

id Tipo: int(11). Llave primaria de la tabla para numerar los estados.

listo Tipo: smallint(2). Indica si la tarea esta en proceso (0), completada (1), cancelada (2) o en espera de recursos (3).

entrada Tipo: text. Descripción de la tarea ejecutada junto con sus parámetros.

tiempo_inicial Tipo: datetime. Fecha y hora en el cual se inicia la tarea. Se cambia cada vez que la tarea se esta ejecutando (i.e. no está esperando recursos).

tiempo_transcurrido Tipo: varchar(255). Tiempo que la tarea ha estado o estuvo en ejecución. Notar que se cambia cada vez que el proceso se detiene, ya sea porque se completa, se cancela o espera recursos.

resultado Tipo: text. Resultado de la tarea ejecutada o mensaje de que la tarea fue cancelada. Blanco en un inicio.

pid Tipo: int(16). Process id del proceso en el nodo main del cluster. Útil para poder cancelar la tarea.

usuario_id Tipo: int(11). Llave foránea que apunta al usuario que inició la tarea.

Esta tabla existe pues es importante para el usuario conocer todos los campos descritos anteriormente, tanto el qué fue lo que ejecutó, sus variables y el tiempo transcurrido, como el resultado de esto. Gracias al campo **pid** también permite cancelar tareas que se están ejecutando en el cluster.

4.4. Diseño de la aplicación

4.4.1. Servidor Frontend

El framework MVC usado para realizar la plataforma web fue Play! framework, el cual usa como lenguaje de programación a Scala.

La aplicación está basada en seis packages:

4.4.1.1. Controllers

En este package se encuentran los controladores de la aplicación. Los objetos que lo forman son cinco:

Secured.scala

Contiene el trait Secured que entrega unos wrappers para realizar verificaciones que se realizan en varias acciones, y así evitar duplicación de código. Tiene las siguientes funciones:

- **IsAuthenticatedR** que verifica si el usuario está logeado y deja en scope el nombre de usuario.
- **IsAdminR** que verifica si el usuario es un administrador y deja en scope el nombre de usuario.
- **canChange(job: Int)** que verifica si el usuario es dueño de la tarea con id **job**.

Login.scala

Contiene el objeto Login que extiende de Controller con el trait Secured. Su labor principal es autenticar a los usuarios que intenten entrar al sistema. Dentro de las acciones que contiene se encuentran:

- La acción **index** que se encarga de verificar si está logeado en el sistema y redireccionar al index de Servicios. Si no, se muestra la vista del login.
- La acción **logout** que redirecciona al index de Login con una nueva variable de sesión, de manera que ya no se encuentre en ella el usuario previamente logeado.
- La acción **authenticate** llamada por el requerimiento POST de la vista del login, que hace un bind del form correspondiente con los datos entregados por el requerimiento y redirecciona al index de Servicios si es que se logró autentificar, agregando el nombre de usuario y si es admin a la variable de sesión, o muestra nuevamente el login con los mensajes de error encontrados si es que no se logró auntentificar.

UsuariosC.scala

Contiene el objeto UsuariosC que extiende de Controller con el trait Secured. Es el encargado de entregar funcionalidades relacionadas con los usuarios, como por ejemplo cambiar contraseña o crear un usuario nuevo. Sólo un administrador puede acceder a ciertas acciones de este controlador. Dentro de las acciones que contiene se encuentran:

- La acción **obtenerUsuario** que muestra la vista para crear un nuevo usuario. Sólo un administrador puede acceder a esta acción.
- La acción **crearUsuario** que recibe los datos del form para crear usuario y muestra la misma vista con errores si es que el usuario ya existe, o crea el nuevo usuario en la base de datos, redireccionando al index de Servicios. Sólo un administrador puede acceder a esta acción.
- La acción **listaUsuarios** que muestra la vista que lista todos los usuarios que no son administradores, y permite promoverlos a administradores. Sólo un administrador puede acceder a esta acción.

- La acción **promover(id: Int)** que promueve al usuario con llave primaria **id** a administrador. Sólo un administrador puede llamar esta acción.
- La acción **cambiarCsña** que muestra una vista que permite introducir una contraseña nueva para el usuario logeado.
- La acción **verificarCambio** que verifica los datos del form para cambiar contraseña y muestra la misma vista con errores si las contraseñas no coincidieron o si la contraseña actual no es correcta, o cambia la contraseña actual si no hubo errores.

Servicios.scala

Contiene el objeto Servicios que extiende de Controller con el trait Secured. Se encarga de entregar las principales labores de la aplicación, como por ejemplo, ejecutar un algoritmo que factorice algún número. Sólo los usuarios autenticados pueden ejecutar estas funciones. Dentro de las acciones que contiene se encuentran:

- La acción **index** que muestra la vista con la lista de servicios.
- La acción **estadoTareas** que muestra la vista con la lista del estado de las tareas ejecutadas por el usuario.
- La acción **supuestos** que muestra la vista con la lista de supuestos que la aplicación es capaz de resolver.
- La acción **showParams** que dado el nombre de un supuesto, muestra una vista que en la cual, se puede ejecutar la resolución del supuesto escogido entregándole los parámetros requeridos.
- La acción **verificarYEjecutar** que dado el nombre de un supuesto, realiza un bind del form del supuesto con los datos del requerimiento, verificando errores y mostrando nuevamente la vista en la que se requieren los parámetros junto con sus errores si es que los hay, o ejecutando la tarea si es que no, mostrando la lista de servicios nuevamente.
- La acción **subirTarea** que muestra una simple vista en la cual se permite al usuario subir su propio script para resolver alguna tarea, o escribiéndolo en el editor online presente en la misma vista, pudiendo usar cualquier resolución de un supuesto presente en el sistema.
- La acción **ejecutarTareaSinArchivo** que mediante código escrito en el editor online, compila y ejecuta la tarea correspondiente.
- La acción **ejecutarTarea** que obtiene el script subido por el usuario, compilando y ejecutando esta tarea.

- La función **compilarYEjecutar(code: String, user: String)** que compila el código **code** y ejecuta esta tarea bajo el usuario **user**. Esta tarea se ejecuta tanto en el lado del servidor si se necesita, como en el cluster. El uso de funciones de la librería estándar de Scala se ejecutan en el servidor y la resolución de los supuestos en el lado del cluster. Para esto, se usa la librería de reflexión de Scala, que puede compilar en runtime un código entregado usando el ClassLoader de la clase en la cual se ejecuta, por lo cual este código podría incluso usar los modelos de la plataforma web, es decir, acceder a la base de datos por ejemplo. Es por esto que mediante una expresión regular sobre el contenido del archivo, se prohibe el uso de packages delicados, como controller, models, views, play, util, y asi proteger el sistema de usuarios malintencionados.

4.4.1.2. Models

Para abstraerse de la base de datos se usó Slick [24], el cual es un FRM (functional relational mapper).

Como hay dos tablas en la base de datos diseñada, también hay dos modelos:

Usuario.scala

Representa la abstracción de la tabla usuario de la base de datos. Posee una clase que refleja una fila de la base de datos, con variables de instancia públicas, del mismo nombre y tipo de cada elemento de la tabla usuario.

También se tiene un objeto Usuarios que representa la tabla en sí. Tiene las columnas **id, nombre_usuario, semilla, password, es_admin**. Posee las siguientes funciones para interactuar con la base de datos:

- **insert(user: Usuario):** inserta el usuario a la base de datos, cuyo password a guardar es el bcrypt [25] del password entregado por el usuario, y el salt generado por la librería jBCrypt con 12 rondas.
- **findByNombre(nombre: String):** busca un usuario cuyo nombre_usuario sea igual a **nombre**.
- **authenticate(hashpass: String, password: String):** verifica si el bcrypt de el pass introducido por el usuario **password**, es igual al encriptado **hashpass**.
- **findNonAdminUsers:** entrega una lista con todos los usuarios que no son administradores.
- **promover(id: Int):** se le otorgan privilegios de administrador al usuario cuyo id sea igual a **id**.
- **nuevaContraseña(user: String, pass: String):** se cambia el password a **pass,** al usuario cuyo nombre es **user**.

Estado.scala

Al igual que usuario, representa la abstracción de la tabla estado de la base de datos, y posee una clase que refleja un elemento de la tabla estado.

También tiene un objeto Estados que representa la tabla, junto con las columnas **id, listo, entrada, tiempo_inicial, tiempo_transcurrido, resultado, pid, usuario_id**. Se pueden encontrar en él las siguientes funciones para interactuar con la base de datos:

- **insert(username: String, entrada: String):** inserta un estado en proceso en la base de datos, cuya entrada es **entrada**, con tiempo inicial el tiempo actual, con tiempo transcurrido 0, sin resultado y cuyo nombre de usuario es **usuario**.
- **findById(id: Int):** dado un entero, se busca un estado cuyo id coincida con **id**.
- **update(id: Int, resultado: String):** modifica el estado cuyo id es igual a **id**, asignándole a este que se encuentra completo, calculando el tiempo transcurrido hasta ese momento y con el resultado entregado.
- **updateForCancelled(id: Int):** modifica el estado cuyo id es igual a **id**, diciendo que se encuentra cancelado, calculando el tiempo total que transcurrió en la tarea y con resultado "Tiempo permitido excedido".
- **updateProcess(id: Int):** modifica el estado cuyo id es igual a **id**, asignando que el estado se encuentra en proceso y reseteando el tiempo inicial de la tarea. La finalidad de este método es llamarlo después de haber esperado por recursos para la tarea, y de esta forma, no considerar este tiempo en la cual el proceso estaba esperando.
- **updateWaiting(id: Int):** modifica el estado cuyo id es igual a **id**, diciéndole que se encuentra esperando recursos y actualizando el tiempo transcurrido hasta ese momento.
- **getEstadosByUsuario(usuario: String):** se entrega una lista de estados del usuario cuyo nombre de usuario es **usuario**.
- **isOwner(usuario: String, id: Int):** verifica si el usuario con nombre de usuario **usuario**, inició una tarea (o posee un estado) cuyo id es **id**.
- **canBeCancelled(id: Int):** verifica si el estado cuyo id sea **id** puede ser cancelado. Si esta procesando o esperando recursos puede ser cancelado, sino no.
- **cancel(id: Int, quien: String):** cancela la tarea asociada al estado cuyo id es igual a **id** realizando un kill -9 al pid del estado con ssh al nodo principal del cluster, y modificando este estado, indicando que se encuentra cancelado, con el tiempo total transcurrido, y cuyo resultado es un mensaje que indica si el proceso fue cancelado por el usuario, o por un administrador. Además, debe parar el thread que se encontraba realizando cálculos o esperando el resultado del cluster y liberar los recursos que el cluster se encontraba usando, para que otra tarea pueda ejecutarse. Ver el objeto Scheduler para más detalles.
- **getTodos:** obtiene una lista de todos los estados existentes. Útil para que el administrador pueda ver y cancelar tareas.

4.4.1.3. Views

En el proyecto se encuentran 9 templates de vistas. Todas las vistas tienen como parámetro implícito (es decir, un parámetro que si no es especificado, es buscado por el compilador) un valor de tipo Flash.

main.scala.html

Template base usado por todas las vistas. Tiene como parámetro el título y el contenido html. También tiene dos parámetros implícitos: una variable flash y un Option de un Form de Play.

Se encarga de obtener los css correspondientes, como el **normalize.css,** que normaliza todos los atributos css que los browser tienen por defecto y el **main.css** que es el css principal de la página.

También coloca los errores globales del form si es que existe un form y si es que posee errores. En el mismo ámbito coloca los mensajes de éxito y de error que se colocan en los flash, de manera de entregar feedback al usuario.

Finalmente, se pone el contenido.

login.scala.html

Vista que tiene como parámetro un form de Play. En su contenido posee un form que pide el nombre de usuario y la contraseña al usuario. Redirecciona al controlador Login.authenticate. También tiene un link que lleva al registro en Registro.index.

listaservicios.scala.html

Además de tener un Flash de parámetro implicito, tiene también una variable de Session, de forma de determinar si el usuario es un admin y así mostrar un link que le permite crear nuevos usuarios y mostrar una lista con todos los usuarios, y mostrar una lista con todos los estados de las tareas.

Tres servicios importantes se dan en la aplicación:

- Encolar ejecución de ataque: muestra un listado de los supuestos que se pueden resolver, de forma de escoger uno para resolverlo con parámetros entregados por el usuario.

- Ver estado de las tareas: muestra un listado de todas las tareas que el usuario ha ejecutado, así como si están listas, en proceso, canceladas o en espera de recursos. Se muestran además los parámetros que el usuario entregó para la ejecución de algún ataque, el tiempo transcurrido o que transcurrió, y el resultado si es que hay.
- Subir tarea personalizada: permite al usuario que suba un script escrito en Scala, que puede usar las funciones que resuelven supuestos definidas en el sistema.

Desde acá también se puede hacer un logout del sistema y cambiar la contraseña de la cuenta.

crearUsuario.scala.html

Tiene como parámetro un form de Play que tiene las restricciones sobre los campos y lo que se puede colocar.

En su contenido hay un form que pide un nombre de usuario. Este form redirecciona al controlador UsuariosC.crearUsuario.

listaUsuarios.scala.html

Posee como parámetro una lista de los usuarios que no son administradores. En el contenido se muestra esta lista en una tabla: el id del usuario, el nombre de usuario y un link que permite promover al usuario correspondiente a administrador, que llama al controlador UsuariosC.promover.

listasupuestos.scala.html

Como parámetro tiene un map que representa todos los supuestos que se pueden resolver en el sistema. Estos los muestra en el contenido del html.

getAlgParams.scala.html

Tiene como parámetros el supuesto correspondiente y el form asociado a ese supuesto.

En el contenido se muestran los campos del supuesto. Cabe destacar que no hay un template distinto por supuesto, sino que cada objeto de tipo supuesto, tiene los input necesarios para el html. El form redirecciona a la acción Servicios.verificarYEjecutar.

getArchivoTarea.scala.html

En este template se muestra un form que pide un archivo del usuario y también se muestra un editor de código online, para poder escribir la tarea a ejecutar en Scala. Se puede subir ya sea un archivo o escribir el código de la tarea en el editor de la página.

estadoTareas.scala.html

Este template tiene como parámetro un boolean y un listado de estados de las tareas correspondientes al usuario logeado si el boolean es falso, o una lista de los estados de todos los usuarios si es verdadero. Esta lista de estados se muestra en una tabla, la cual dice si una tarea esta en proceso, completa, cancelada o en espera de recursos. También muestra la entrada con la cual se ejecutó o ejecuta el algoritmo, junto con el resultado si es que existe y el tiempo transcurrido o que transcurrió.

cambiarPassword.scala.html

Este template tiene como parámetro un form de Play que representa la entrada del usuario. En la vista se tiene un form que pide la contraseña del usuario, así como la nueva contraseña junto con la repetición de ésta. El form redirecciona a la acción UsuariosC.verificarCambio.

4.4.1.4. Util

Package que contiene principalmente cuatro objetos: Global.scala, Utils.scala, Scheduler.scala y ProcessJobs.scala.

Global.scala

Objeto que contiene un sistema de actores que sólo es relevante para el uso del timer cancelable usado en ProcessJobs. También, es el encargado de inicializar el Scheduler al comienzo de la aplicación.

ProcessJobs.scala

Objeto que tiene una función importante para la ejecución de los procesos. Esta función, llamada **runThread,** dado un usuario, una entrada, un valor por nombre resultado, el Option de una Tarea compilada con la librería de reflexiones de Scala y un Option de un

Supuesto, crea un Thread que básicamente inserta un nuevo estado en proceso con los parámetros necesarios, agrega el id de este estado en la lista de tareas en ejecución y con la entrada dada ejecuta la computación del resultado de la tarea, ya sea una escrita por el usuario o una resolución de un supuesto implementado en el sistema y luego modifica el estado creado anteriormente para que contenga el resultado y se encuentre “completo”. Finalmente quita de la lista de tareas en ejecución la tarea con el id del estado insertado.

Sin embargo, lo anterior no es suficiente, pues al estar ejecutando código del cliente, y por error o no, es posible que este código use demasiado tiempo e incluso podría no terminar (loop infinito, con una recursión o un while true). Es por esto que se implementa una solución para este problema, tomando en cuenta que la ejecución de los algoritmos que resuelven los supuestos también pueden ocupar mucho tiempo. Con esto, la solución fue la siguiente: primero se crea un timer, que es asignado al objeto Tarea creado por el usuario (y este timer es implícito) y antes de realizar la computación del resultado, se inicia el timer. Entonces, cada vez que el usuario llama a una función que resuelve un supuesto computacional, dentro de éste se detiene el timer, se resuelve el supuesto obteniendo el resultado y se reinicia el timer nuevamente. Si se le acaba el tiempo al timer, entonces se mata al thread que estaba ejecutando la tarea del usuario. De esta forma, aseguramos que el usuario no pueda ocupar demasiado tiempo de procesamiento en el servidor y el cluster se puede demorar lo que sea necesario.

Para realizar lo anterior, se implementó una clase MutableCancellable que tiene el id del estado creado, una variable Cancellable (que es un timer que se puede cancelar implementado en la librería de actores de Akka [26]), una variable que indica el tiempo que falta y otra que indica en qué momento se llega al tiempo límite. Su método cancel cancela el timer Cancellable y su método start crea un nuevo timer Cancellable que ejecuta una función despues de un tiempo que es dado. Esta función, implementada en el objeto Utils, mata el thread y asigna el estado de la tarea como “cancelado”.

Scheduler.scala

Objeto que tiene las funciones y los objetos necesarios para el manejo de los cores del cluster. No se usó una herramienta que realiza esto, debido a la complejidad que esto podría tener y al enfoque que posee este trabajo.

A grandes rasgos esto funciona de la siguiente manera: se tiene una lista de tareas, una lista de recursos que son IPs de los nodos, una cola de los Supuestos que iban pidiendo los recursos, la cantidad de cores totales del cluster y una cantidad de cores disponibles. Cuando la resolución de un supuesto pide recursos, se agrega a la cola y se verifica si hay cores disponibles para él. Si no hay, queda esperando hasta que existan. Cuando hayan disponibles, se le asignan a la ejecución. Cuando termina de ejecutarse la resolución de un supuestos, este libera los recursos, quitando al supuesto de la lista de tareas, re-asignando los recursos tomados por el Supuesto y finalmente avisarle al resto de Supuestos que salió uno, para que tome los recursos si hay suficientes. Como se puede ver, este Scheduler es del tipo FIFO, pidiendo los recursos y después liberandolos cuando termine. Se quiso implementar de esta manera pues era el más simple.

Para entrar en más detalles, se explicará cada uno de los campos de este objeto:

- **tareas:** HashMap cuya llave es un entero que representa el id de un estado y cuyo valor es un par (Option[Supuesto], Thread). Util para cancelar procesos: si un usuario o administrador quiere cancelar una tarea, entonces se mata el thread de la tarea, se liberan los recursos y se mata el proceso distribuido en el cluster, mediante el campo pid del estado.

- **resources:** Lista sincronizada (para manejar concurrencia) y mutable que contiene las IPs de todos los nodos disponibles del cluster. Inicialmente se encuentran repetidas si es que un nodo tiene más de un core. Por ejemplo, si aparece una IP repetida 4 veces, entonces ese nodo posee 4 cores disponibles.

- **processQueue:** Cola sincronizada y mutable que contiene los Supuestos que están esperando recursos.

- **coresTotales:** Valor entero que tiene la cantidad de cores totales del cluster. Para esto lee un archivos hosts que contiene las IPs de todos los nodos del cluster, junto con la cantidad de cores.

- **hostList(res: List[String]):** Función de utilidad que toma la lista de recursos y la parsea como un string con el siguiente formato: Ip1:cores1,...,Ipn:coresn. Ip1 representa la ip de un nodo del cluster y cores1 representa la cantidad de cores que se van a usar de aquel nodo. Útil para sshRemote del trait Supuesto, pues sirve como parámetro para mpirun.

- **coresDisponibles:** Variable entera que representa la cantidad de cores disponibles para usar.

- **allocateResources(cantidad: Int):** Toma recursos, disminuyendo la cantidad de cores disponibles en **cantidad** y quitando **cantidad** IPs de la lista resources, retornando las IPs quitadas.

- **takeResources(sup: Supuesto):** Función que se ejecuta antes de esperar el resultado en la resolución de un supuesto. Agrega un nuevo par al HashMap tareas, con el id del Supuesto, el Supuesto y el thread actual, se agrega el supuesto a processQueue, se cambia el estado del Supuesto a esperando recursos, actualizando el tiempo transcurrido, y si no hay cores disponibles para el supuesto o ya hay otro Supuesto al principio de la cola, entonces se queda esperando. Cuando termine un Supuesto y despierte a todas las tareas esperando usando un monitor, se verifica nuevamente la condición. Si hay cores disponibles, y es el primero de la cola, entonces modifica nuevamente el estado correspondiente para que este diga que se encuentra en proceso, actualizando el tiempo inicial (y así no medir el tiempo que estuvo esperando). Finalmente, se saca de la cola y se le asignan los recursos al Supuesto. Todo esto se hace sincronizando en processQueue, para evitar acceso concurrente a los recursos.

- **freeResources(sup: Supuesto):** Función que se ejecuta después de esperar el resultado en la resolución de un supuesto. Sincronizadamente en processQueue,

se saca el Supuesto de la lista de tareas, se reponen los recursos pedidos a la lista resources, se repone la cantidad de cores disponibles y finalmente despierta a todos los procesos esperando por recursos.

Util.scala

Objeto que posee funciones de utilidad. Estas funciones son:

- **periodToString(period: Period):** Función encargada de representar un Period de la librería de joda-time como un String con el formato: A Año(s), B Mese(s), C Semana(s), D Día(s), E Hora(s), F Minuto(s), G Segundo(s).
- **stringToPeriod(s: String):** Función encargada de parsear un String, para poder representarlo como Period de la librería joda-time.
- **timeToShow(e: Estado):** Función encargada de mostrar el tiempo transcurrido de forma correcta, dependiendo de si la tarea esta en proceso, completa, cancelada o esperando recursos. Si esta en proceso, entonces se calcula el tiempo que lleva hasta ese momento. Si esta completa, cancelada o esperando recursos, entonces sólo se muestra el tiempo transcurrido que se encuentra en **e**.
- **calculateTime(tiempo_inicial: DateTime, tiempo_transcurrido: String):** Calcula el tiempo que ha pasado hasta el momento, tomando en cuenta el **tiempo_transcurrido** y el **tiempo_inicial**. Informalmente retorna **tiempo_transcurrido + (tiempoahora - tiempo_inicial)**.

4.4.1.5. Supuestos.base

Este package cuenta con cuatro archivos importantes, que representan la base para la capacidad de agregar resoluciones de nuevos supuestos de manera simple.

Tarea.scala

Trait que representa una tarea subida por un usuario. Tiene una variable Option de MutableCancellable que por defecto es None, un método descripción que indica la descripción de la tarea y puede ser sobreescrita por el usuario. Otro método a implementar es el método run que es el que se encarga de realizar las computaciones del usuario.

HtmlFields.scala

El trait HtmlField en este archivo representa un input de un form de HTML. Tiene un campo nombre para asignarle un nombre al input y un campo que es una función de cuatro variables que representan: el id del elemento, el nombre del elemento, el valor del elemento (el cual es un Option pues puede que no tenga valor) y argumentos extras para el elemento, el cual retorna un objeto de tipo Html. Esta función se usa con el helper.input de los templates de Play [27] y de esta forma poder personalizar los elementos, usando siempre el helper input de la librería de templates, y así tener un template genérico para cualquier lista de HtmlFields.

Se encuentran actualmente implementadas las clases InputText que representa un input simple de texto y la clase Select que representa un input de selección de una lista.

Validations.scala

El trait Validation representa una validación a hacerse a los campos del Form de los supuestos, para así poder agregar una restricción (constraint) nueva de forma fácil, usando las ya implementadas por Play. Estas restricciones (o constraints) son del tipo Mapping[_].

Algunas clases implementadas son:

- Text que indica que el campo tiene que ser un texto.
- BigDec que indica que el campo tiene que ser un decimal grande.
- BigInteger que indica que el campo tiene que ser un entero de precision arbitraria.
- Number que indica que el campo tiene que ser un entero.
- Hex que indica que el campo tiene que ser un valor en hexadecimal.
- Cores que indica que el campo tiene que ser un entero y menor a la cantidad de cores totales del cluster.

Supuesto.scala

Se encuentran en este archivo tres traits importantes al momento de agregar un nuevo supuesto, ya sea usando supuestos ya implementados o ejecutando solamente un algoritmo del cluster.

El trait ConstExtrSupuesto representa el constructor y extractor del objeto que posee los parámetros de la resolución del supuesto. Se debe implementar el constructor. El

constructor es un método que dado una cierta cantidad de parámetros retorna una instancia del objeto requerido. Es usado para poder generar un Form de Play de tamaño arbitrario con los parámetros del supuesto.

El trait InputSupuesto modela principalmente un Form de Play para que el usuario pueda introducir los parámetros del supuesto escogido. Está hecho de forma de ser genérico, es decir, el usuario no tiene que implementarlo explícitamente, sino definir los métodos requeridos. Tiene los siguientes métodos:

- **nombre:** Representa el nombre del supuesto del cual se pedirán sus parámetros. Debe ser implementado.
- **f:** Contiene el ConstExtrSupuesto necesario para que pueda ser usado por el generador de Forms. Debe ser implementado.
- **fields:** Es la lista de objetos del tipo FieldWithValidation, es decir, del tipo mixing entre HtmlFields y Validation.
- **form:** Es el generador de Form para un supuesto cualquiera. Se basa inicialmente en la función mapping de Play que pide como parámetros varios pares (String, Mapping[_]), en donde Mapping[_] representa una restricción en los campos (por ejemplo, todos los strings que sean mails), y el constructor junto con el extractor del objeto a formar al hacer el bind con los datos del usuario. Sin embargo, esta función tiene limitaciones: hay un overload de esta función para cada número de pares, es decir, una función mapping con un par, con dos pares, con tres...así hasta llegar a 21 pares. Por lo tanto no puede ser genérica. Es por esto, que el mapping que retorna para crear el Form, es un mapping de un par, en donde el segundo par, es el mapping de otro par; así hasta llegar a un sólo elemento y crear el Form. Todo esto usando la lista de fields.

El trait Supuesto representa la ejecución de la resolución del supuesto elegido y posee los parámetros necesarios para su resolución. Posee los siguientes métodos:

- **resources:** Lista de los recursos tomados por esta ejecución. Es una lista de IPs, en donde cada elemento de la lista representa un nodo en donde se va a ejecutar la tarea. Pueden haber IPs repetidas, lo cual significa que toma más de un core o núcleo en el nodo correspondiente.
- **idEstado:** Representa el id del Estado en la base de datos, con el cual está asociado esta ejecución.
- **cores:** Entero que indica la cantidad de cores que toma esta ejecución. Tiene que ser implementado o puede ser una valor en la instancia de una clase.
- **params:** Posee el valor, como cadenas de carácteres, de cada uno de los parámetros del supuesto. Tiene que ser implementado.

- **sshRemote:** Retorna una secuencia de strings que representan la ejecución de ssh, con una llave privada, hacia el nodo principal del cluster que ejecuta el algoritmo en los distintos nodos que se le asignaron a la tarea (resources).
- **entrada:** Dado un InputSupuesto, entrega un string que describe el nombre del supuesto, y el nombre de los parámetros con sus valores.
- **process:** Retorna un proceso que representa la ejecución del algoritmo en cuestión. Puede ser ejecutado y debe ser implementado.
- **execute:** Dado un parámetro por nombre que retorna un string y un Option de un MutableCancellable, se cancela el timer del MutableCancellable y se guarda el id del estado en idEstado que viene en él, se piden los recursos necesarios, se ejecuta el bloque por nombre, se liberan los recursos, se reanuda el timer y se retorna el resultado de este bloque por nombre. Representa la ejecución de un bloque, en donde no se gasta el tiempo destinado al usuario. Cabe notar, que se pueden tener estos bloques de execute anidados, por ejemplo, dentro de un bloque de execute, tener una ejecución a Factorizar, realizar cierto cálculo y tener otra ejecución a Factorizar (al ejecutar Factorizar, se tiene otro bloque execute dentro). Esto en un principio causaba que se reiniciara el timer antes de terminar la ejecución del primer execute, pues en el execute del primer Factorizar, primero se para el timer y después se vuelve a iniciar, sin considerar que el execute de más afuera aún se estaba ejecutando. Por ende, en MutableCancellable fue necesario agregar un contador inicializado en 1, en donde cada vez que se hacía start, se restaba 1 y si el contador era 0, se hacía efectivo el inicio del timer, y cuando se realizaba un cancel se agregaba 1 al contador, y se detenia el timer cuando el contador fuese 1. De esta forma, el problema anterior se solucionó.
- **run:** Dado un parámetro implicito de un Option de un MutableCancellable, se hace un execute con la ejecución de process. Puede ser reimplementado por el usuario.
- **ejecutarAlg:** Dado un nombre de usuario, y un InputSupuesto, ejecuta la resolución del supuesto llamando a runThread. Cabe destacar que si a runThread no se le entrega una Tarea, el timer no avanza, pues en runThread se inicia el timer, y luego se realiza la computación del resultado, y en este caso, la computación del resultado es un execute que apaga el timer en el inicio.

4.4.1.6. Supuestos

Se encuentran implementados en el sistema 5 supuestos, los cuales se pueden resolver:

Factorización

Representa el supuesto de factorizar. Al igual que en Supuesto.scala hay tres clases/objetos importantes que se derivan de aquellos en Supuesto.scala.

La primera clase Factorizar, representa la tarea de factorizar un número. Posee dos parámetros: un número entero de tamaño arbitrario y la cantidad de núcleos a ocupar por el algoritmo. El método process es implementado usando sshRemote, el cual ejecuta un binario llamado “trial_division” que factoriza usando fuerza bruta con el número entregado. También se reimplementa el método run para formatear el resultado entregado por el proceso.

El objeto FactConstExtr que extiende de ConstExtrSupuesto, implementa el constructor de Factorizar dado dos tipos paramétricos (esto por los pares de los pares descrito anteriormente).

Finalmente el objeto Factorizacion que extiende de InputSupuesto, el cual implementa **nombre** y **f** entregándole el objeto FactConstExtr. También, se implementa el método **fields,** el cual es una lista simple de dos elementos: un objeto InputTextWithBigDec y un objeto InputTextWithNumber. Estas últimas clases son simplemente mezclas entre un trait de tipo InputText, con otro trait BigDec y Number respectivamente, que extienden de Validation.

Debido a que esto es análogo para todos los supuestos, sólo se describirán los parámetros de la clase que extiende de Supuesto.

Logaritmo Discreto

Representa el supuesto de la dificultad de encontrar el logaritmo discreto. Esto está sobre el grupo de enteros módulo p con respecto a la multiplicación, $\mathbb{Z}_p^*$.

La clase DL representa la tarea de encontrar el logaritmo discreto de un número. Como parámetro tiene un número y, un primo p y un generador g (todos estos de tipo BigInteger), junto con la cantidad de núcleos a usar, para encontrar el logaritmo discreto de y. Al igual que Factorizar, el método process se implementa usando sshRemote con el binario de logaritmo discreto usando fuerza bruta.

Logaritmo Discreto en curvas elípticas

Representa el supuesto de la dificultad de encontrar logaritmos discretos sobre el grupo de curvas elípticas sobre $\mathbb{F}_p$, para un primo p.

Análogo a la clase DL, la clase DLEC representa la tarea de encontrar el logaritmo discreto de un par, sobre las curvas elípticas. Se le pide el punto $P = (p_x, p_y)$ al cual determinar el logaritmo discreto, las constantes A y B de la curva elíptica, el primo p sobre el cual está definido $\mathbb{F}_p$, y el generador $G = (g_x, g_y)$. Nuevamente, el método process se implementa usando sshRemote, poniendo el binario correspondiente junto con los parámetros entregados.

Colisiones de Hash

Representa el supuesto de la dificultad de encontrar colisiones sobre funciones de hash.

La clase EncontrarColision refleja la tarea de encontrar una colisión sobre un hash dado, con una cantidad de bytes de truncación dada y con la cantidad de bytes en cero que tienen que tener los puntos distinguidos dada, además de la cantidad de núcleos a usar. Si la cantidad de bytes a truncar es 0, entonces no se trunca el resultado de la función de hash. Al igual que los anteriores, process usa sshRemote con los parámetros que fueron pasados.

En este caso es importante destacar que se crea un Select sólo pasándole el nombre del select y una lista de los valores a mostrar.

Preimagenes de Hash

Representa el supuesto de la dificultad de encontrar una preimagen en una función de hash.

La clase EncontrarPreimagen se asimila a la tarea de encontrar una preimagen sobre un hash dado, una imagen del hash dada, una cantidad de bytes de truncación dada y la cantidad de núcleos a usar. Como con los anteriores, process se implementa usando sshRemote junto con el binario y los parámetros correspondientes.

4.4.2. Cluster AWS

En el cluster AWS se implementan algoritmos para resolver los supuestos computacionales que se han ido mencionando usando MPI, pues se está trabajando sobre un entorno distribuido. Para esto se instalaron las librerías necesarias: NTL [28], GMP [29], MIRACL [30], CryptoPP [31] y MPICH-2 [48].

Los algoritmos implementados para Factorización y Logaritmo Discreto ya fueron descritos anteriormente y corresponden a la resolución de estos problemas mediante fuerza bruta. Esto debido a la sencillez de implementación que estos algoritmos entregan. En un trabajo posterior sería posible agregar otras implementaciones de algoritmos, como por ejemplo, Number Field Sieve para factorización, el cual es más rápido que Trial Division para números grandes.

En cuanto a poder encontrar colisiones sobre funciones de hash, se implementa un ataque tipo cumpleaños distribuido. Finalmente, para encontrar preimágenes en funciones de hash, también se implementa un algoritmo de fuerza bruta.

A continuación se dará una descripción pequeña sobre las librerías usadas y sobre la forma de distribuir de cada uno de los algoritmos:

Factorización

Para implementar la factorización por fuerza bruta o Trial Division, se usó además de MPI, la librería NTL [28] que entrega operaciones y funciones sobre diversas estructuras algebráicas, como por ejemplo $\mathbb{Z}$ y $\mathbb{Z}_n$. NTL se puede usar junto con GMP [29] que es una librería que permite el uso de enteros grandes de forma eficiente.

Para distribuir la factorización se hace lo siguiente: dado n el número a factorizar y dado s procesos, el proceso $i \in \{0, ..., s-1\}$ busca factores primos (mediante el test de Miller-Rabin [8]) por trial division en el intervalo $\left[\lfloor\sqrt{n}\rfloor\frac{i}{s} + 1,\ \lfloor\sqrt{n}\rfloor\frac{i+1}{s}\right]$, junto con la multiplicidad del factor y se guardan estos factores en un vector. Luego de recorrer todo el intervalo, manda los factores encontrados al proceso $s-1$. Este proceso $s-1$ es el encargado de recibir todos los factores de todos los procesadores y de entregar el resultado final.

Logaritmo Discreto sobre $\mathbb{Z}_p^*$ y sobre curvas elípticas

La resolución del Logaritmo discreto también fue hecho usando el algoritmo de fuerza bruta, tanto sobre $\mathbb{Z}_p$, como sobre curvas elípticas pues el problema es análogo. Para implementar el algoritmo de fuerza bruta sobre $\mathbb{Z}_p$ con p primo, se usó la librería NTL junto con GMP, y para implementar el algoritmo sobre curvas elípticas, se usó NTL, GMP y MIRACL [30] que es una librería que permite operaciones sobre el grupo de curvas elípticas sobre $\mathbb{F}_p$.

La distribución del trabajo sobre ambos algoritmos es análogo: dado un elemento y a calcular logaritmo discreto, un generador g, un primo p (y las constantes $A, B \in \mathbb{F}_p$ para determinar la curva elíptica), y dados s procesos, cada proceso $i \in \{0, ..., s-1\}$ busca el logaritmo discreto en el rango $A = \left[m\frac{i}{s},\ m\frac{i+1}{s} - 1\right]$ verificando si existe algún $k \in A$ tal que $g^k = y$, en donde m es el tamaño del grupo con el cual se está trabajando. Para el caso de $\mathbb{Z}_p^*$ el tamaño es simplemente $p-1$, sin embargo para el caso de una curva elíptica esto no es así, por lo cual se tiene que recurrir a un algoritmo de orden polinomial que calcula el tamaño de la curva elíptica (schoof [32]). Se asume, por otro lado, que el elemento g es efectivamente un generador, pues para verificar que es un generador hace falta factorizar el orden del grupo.

Cuando un proceso encuentra el logaritmo discreto, le avisa a todo el resto de los procesos que lo ha encontrado para que no busque más, y además le manda el logaritmo discreto encontrado al proceso $s-1$ el cual reporta el resultado.

Colisiones de Hash

Para encontrar colisiones de hash se usa un ataque tipo cumpleaños distribuido que usa puntos distinguidos, el cual fue descrito anteriormente. También, debido a la diversa

cantidad de funciones de hash, se ha hecho extensible y genérico en el algoritmo de hash a usar, siempre y cuando se implemente una clase que tiene tres métodos estáticos: **DIGESTSIZE** que retorna la cantidad de bytes que posee la función de hash, **TruncatedHash** que retorna el hash truncado a una cantidad dada de bytes y **Name** que retorna el nombre del algoritmo.

Para obtener los algoritmos de hash más conocidos se hizo uso de una librería criptográfica llamada CryptoPP [31], que dentro de sus funciones se encuentra la posibilidad de calcular diversas funciones de hash, entre ellas por ejemplo, MD4, MD5, SHA1, SHA-256, etc. Denominaremos H_n como la función H restringida a n bytes.

El algoritmo distribuido es como sigue: dado un algoritmo de hash H, la cantidad de bytes a truncarlo n, la cantidad de ceros p de los puntos distinguidos, y dados $s+1$ procesos, el proceso $i \in \{0, ..., s-1\}$ se encarga de encontrar puntos distinguidos, es decir, se escoge al azar $x \in \{0,1\}^{8n}$, y se calcula $H_n(x), H_n^2(x), H_n^3(x), ...$ hasta que se encuentra un $k \in \mathbb{N}$ tal que $H_n^k(x)$ sea un punto distinguido, el cual manda a un proceso s junto con k y x, y escoge nuevamente un nuevo x al azar. El proceso s recibe los puntos distinguidos, los cuales los guarda en una lista junto con su x y k que los generaron. Cuando se repite un punto distinguido, entonces puede que haya encontrado otro camino por el cual se llega al mismo punto distinguido, y por ende puede encontrar una colisión. Si se encuentra, entonces le avisa a todos los demás procesos que paren y retorna el resultado.

Preimágenes de Hash

Para encontrar preimágenes en funciones de hash simplemente se usa un ataque de fuerza bruta. Al igual que con las colisiones de hash, es posible usar cualquier función de hash mientras se implementen los métodos mencionados anteriormente. Se usa la librería CryptoPP para obtener las funciones de hash.

La descripción del algoritmo es la siguiente: dado un algoritmo de hash H, la cantidad de bytes a truncarlo n, una imagen y del hash H_n y dados s procesos, el proceso $i \in \{0, ..., s-1\}$ genera al azar una cadena x de tamaño entre 1 y 512, calcula $H_n(x)$ y verifica si esto es igual a la imagen y. Si no lo es, se genera otra cadena nueva y se continúa el proceso, y si lo es, avisa al resto de procesos que una preimagen fue encontrada y se manda la preimagen al proceso $s-1$ el cual reporta el resultado.

Las implementaciones de estos algoritmos se sitúan en la unidad NFS para que se encuentren disponibles en todos los nodos. Además de los algoritmos, se necesitaba para el uso de MPI, un archivo de hosts que contuviese todas las IPs de los nodos con la cantidad de núcleos totales de cada uno, y de esta forma no ejecutar una tarea en un nodo o vCPU que ya se encuentra en uso (ver Scheduler). Para esto se creó un Daemon de linux que agrega la IP de la instancia junto con la cantidad de núcleos a un archivo ubicado en la unidad NFS al inicio del sistema, y que borra la IP del archivo al parar la instancia. De esta forma, y creando un AMI personalizado, es posible tener un cluster dinámico que

crece o disminuye de tamaño, dependiendo de la cantidad de instancias corriendo con este AMI.

4.4.3. Uso de los scripts de Usuario

En las secciones anteriores se mostraron detalles de implementación del sistema desarrollado. Una de las funcionalidades implementadas, es la de permitir al usuario ejecutar en el servidor su propio código Scala que puede usar las funciones implementadas en el sistema para resolver supuestos. Se explicará cómo debe ser el script que el usuario puede ejecutar en el servidor.

El script que el usuario puede subir debe implementar un método *run* y opcionalmente un método *descripción* y representa una instancia del Trait Tarea mencionado anteriormente. El método descripción, si es implementado, debe sobrescribir con un *override* al método de su clase padre Tarea, y debe retornar un String cualquiera, preferentemente uno que describa los parámetros y el objetivo del script, por ejemplo, decir que la tarea intenta encontrar la clave privada de RSA, para ciertos parámetros.

El método run debe ser implementado obligatoriamente, pues es el método que realiza las computaciones del usuario y tiene que retornar un String con el resultado.

A continuación se muestra un pequeño ejemplo de un script que resuelve RSA para unos parámetros pequeños:

```
1  override def descripcion = "RSA con n=3337 y e=79"
2
3  def run = {
4          import scala.BigInt
5          import supuestos._
6          val n = BigInt("3337")
7          val e = BigInt(79)
8
9          val fact = Factorizar(n, 2).run
10
11         val pattern = "([0-9]+)".r
12         val seq = (for(m <- pattern.findAllMatchIn(fact)) yield m.group(1)).
               toList
13         val p = BigInt(seq(0))
14         val q = BigInt(seq(1))
15         e.modInverse((p-1)*(q-1)).toString
16 }
```

Línea 1: muestra la definición del método descripción.

Línea 4: importa la librería estandar de Scala para manejar enteros de precisión arbitraria.

Línea 5: importa las clases que permiten resolver ciertos supuestos. Son aquellos descritos en la subsección de Supuestos anteriormente.

Líneas 6 y 7: definen la clave pública de RSA.

Línea 9: factoriza instancia la clase que resuelve factorización y pide que lo resuelva para n.

Líneas 11, 12, 13 y 14: transforman el String que contenía la factorización de n, usando una expresión regular, a dos enteros de precisión arbitraria.

Línea 15: calcula la clave privada de esta instancia de RSA y la retorna como String.

Como fue dicho anteriormente, el tiempo que el servidor le da al usuario para realizar sus cálculos está limitado por 1 minuto (sin considerar las llamadas a las funciones que resuelven supuestos), para evitar el abuso indiscriminado de esta característica. Este tiempo fue escogido principalmente para poder testear esta funcionalidad, sin embargo, es muy probable que aumente para cubrir las necesidades de los usuarios.

Para conocer más sobre la sintaxis, funcionalidades y características de Scala ver [49].

4.4.4. Extensibilidad de Supuestos

Para que se puedan agregar al sistema nuevos supuestos o problemas computacionales a resolver, se tienen que implementar una clase y dos objetos.

La clase a implementar debe ser una sub-clase de Supuesto, y representa la ejecución del algoritmo criptoanalítico. Esta clase debe tener implementados dos métodos: **params** que retorna una lista con los parámetros de la ejecución, y **process** que retorna un Process de Scala, que representa la ejecución del binario correspondiente al algoritmo implementado.

Se tiene que implementar además, un objeto constructor, es decir, un objeto con un método **apply**, con dos parámetros de tipos distintos, y que retorna una instancia de la clase mencionada anteriormente. Estos tipos distintos, representan un par de pares, y está implementado de esta forma debido a las limitaciones con los forms de Scala (ver package Supuesto.base).

Finalmente, se tiene que implementar un objeto que representa el form mostrado al usuario. Este objeto debe tener tres valores: un **nombre** del problema a resolver, un valor **f** que tiene el objeto constructor anterior, y un campo **fields** que es una lista de objetos en donde cada uno de los objetos representa el tipo de campo en html (input text, textarea, select, etc.) junto con una validación a hacerse (si lo introducido debe ser un número, un flotante, texto no vacio, etc.).

4.5. Diseño de la Interfaz de Usuario

La interfaz de usuario, como fue descrita en términos de las vistas del framework Play, cuenta con un login, la que consulta por un usuario y una contraseña. De esta manera, podemos limitar el acceso a esta aplicación y evitar un porcentaje de usuarios malintencionados o que no posean el perfil necesario para poder ocupar el sistema. Es por esto que un administrador es el que debe crear un usuario si es que se le va a dar acceso a la persona que lo solicita.

Si un usuario se autentifica, se mostrará una lista de servicios disponible en el sistema. Entre los servicios se encuentran: ejecutar un ataque, subir un ataque personalizado y ver el estado de las tareas ejecutadas.

Al querer ejecutar un ataque, se le mostrará al usuario una lista de ataques que puede ejecutar en el sistema, en la cual cada uno tendrá una serie de parámetros. Por ejemplo, si se quisiera resolver logaritmo discreto, se le pedirá al usuario el elemento a calcular el logaritmo discreto, junto con el generador del grupo, y el primo p si es que se trabaja sobre $\mathbb{Z}_p^*$.

Cuando se quiere subir un ataque personalizado, el usuario debe escribir un script en Scala que puede usar o no los ataques ya implementados en el sistema. Este Script no puede importar librerías del framework ni del sistema, salvo el package en donde están implementados los ataques. El tiempo de usuario que se le otorga al script es un tiempo finito, para evitar sobrecargar el servidor y para evitar un mal uso de este, como loops infinitos.

Por último, si quiere ver el estado de las tareas, se le mostrará al usuario una tabla con todos los ataques que ha ejecutado y que pueden estar en proceso, completados o cancelados, junto con los parámetros entregados por la tarea y el resultado si es que tiene. Una tarea estará en proceso si aún se está ejecutando en el sistema, ya sea realizando operaciones del usuario o esperando que el cluster resuelva alguno de los supuestos o esquemas implementados. Una tarea está completada si es que la ejecución del ataque ya ha terminado y un resultado ha sido mostrado. Una tarea está esperando recursos si no hay núcleos disponibles en el cluster para ejecutar la tarea. Por último, una tarea está cancelada si se ha sobrepasado el tiempo de ejecución del usuario.

Cabe destacar que el usuario no sabe que por debajo se está requiriendo el poder de cómputo del cluster, así que este sistema es de fácil adaptación a cualquier arquitectura en el back-end (sólo que hay que implementar o adaptar los algoritmos en este back-end).

En un trabajo futuro se podría hacer que, además de la existencia de la página web que permite ejecutar tareas personalizadas usando la API de resolución de supuestos, se permita ejecutar algún ataque personalizado sin la necesidad de tener que estar en la página web, sino con una especie de servicio web, donde el usuario incluso podría elegir alguno de los lenguajes que estarían disponibles.

4.6. Argumentación de la solución

Los requisitos de la solución, de forma muy breve, son los siguientes:

1. Un usuario permitido podrá ejecutar un ataque criptoanalítico abstrayéndose de cómo está implementado.
2. Un usuario permitido podrá crear un ataque criptoanalítico usando algoritmos ya implementados.
3. La solución tiene que ser de fácil acceso.
4. La solución debiese ser extensible.

Para el primer requisito, vemos que la solución presentada permite ejecutar ciertos ataques criptoanalíticos, como factorizar, encontrar el logaritmo discreto, encontrar colisiones de hash y encontrar preimágenes en funciones de hash, en donde no es necesario que el usuario sepa su implementación específica, ni es necesario que use su computador.

Con respecto al segundo punto, el sistema presentado permite justamente que el usuario pueda subir un script en Scala que puede ocupar los algoritmos ya implementados en el sistema, como factorizar, etc. De esta forma el usuario podría subir un script con el cual sea posible resolver RSA, dado que es posible factorizar.

El cuarto punto también está disponible, pues para un desarrollador que pudiese estar involucrado con el sistema, puede incluir un nuevo ataque criptoanalítico de forma muy sencilla si es que se encuentra disponible la implementación de la resolución de los supuestos en que se basa este nuevo ataque. Usando el mismo ejemplo anterior, incluir un ataque al algoritmo de encriptación RSA es muy sencillo, dado que existe en el sistema la capacidad para factorizar un número.

Finalmente, el tercer punto se cumple, pero con ciertas limitaciones. Si bien el internet hoy en día, posee un acceso fácil (por ende la solución desarrollada), no siempre se podrá tener acceso a los servicios importantes del sistema de manera inmediata para un usuario nuevo, pues primero un administrador o encargado del sistema tiene que otorgar este acceso. Sin embargo, debido a que el sistema es propenso a ser usado de forma malintencionada y no con fines académicos y/o educacionales, esta restricción tiene que existir.

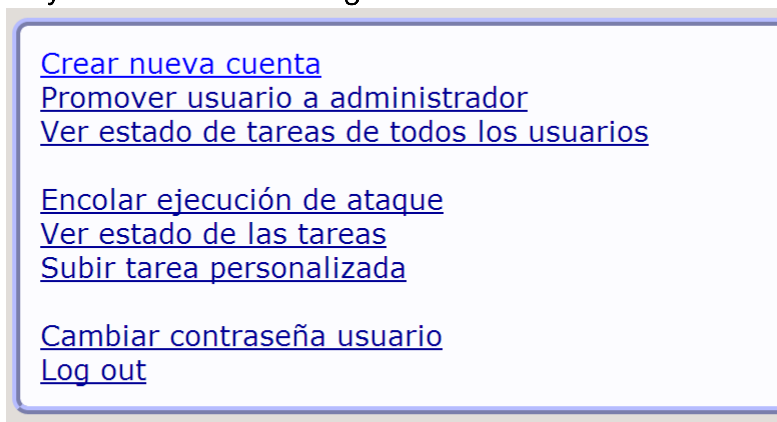
En el siguiente capítulo, se mostrará con un caso de uso, el grado con el cual los objetivos fueron logrados.

Capítulo 5

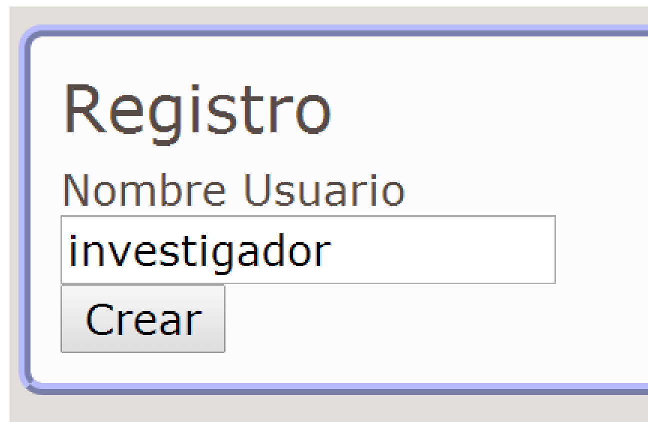
Validación de la Solución

Un investigador de la Universidad de Chile quiere realizar una medición práctica del tiempo que toma factorizar algunos números de una cantidad determinada de bits. Sin embargo, el investigador no tiene a su disposición una máquina dedicada para esto, ni un algoritmo implementado que tome un tiempo razonable en terminar. Es así como este investigador se entera de la existencia del sistema Cryptfront que permite resolver distribuidamente supuestos criptográficos y entre ellos se encuentra resolución del supuesto por el cual está interesado.

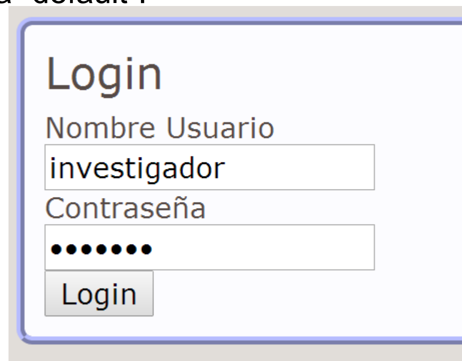
El investigador entonces, se comunica con el encargado de este sistema, comentandole su problema. Con esto, el encargado decide crearle una cuenta para que pueda acceder al sistema y continuar su investigación:



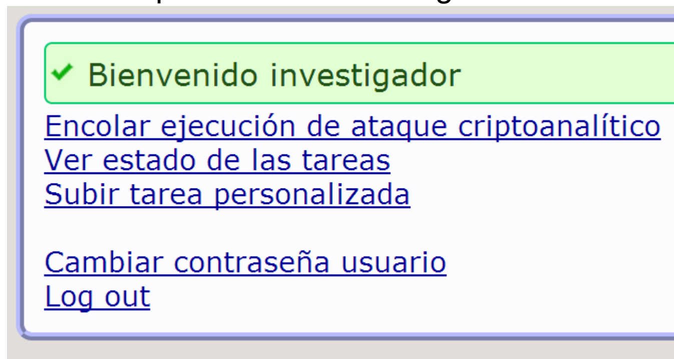
Se le coloca al investigador un nombre de usuario que él desee:



Ahora, dado que el investigador tiene su nueva cuenta, puede hacer *log in* en el sistema con una contraseña “default”:



Lo primero que ve el usuario al entrar al sistema, es la lista de servicios que puede realizar, entre ellas, el servicio por el cual el investigador está interesado:



Sin embargo, debido a la debilidad de la contraseña entregada, es extremadamente recomendado que la cambie inmediatamente. Así el investigador cambia su contraseña a una más segura:

Cambiar Contraseña

Contraseña Actual

•••••••

Nueva Contraseña

••••••••••••••

Repetir Contraseña

••••••••••••••

Cambiar

El sistema le informa que la contraseña fue cambiada con éxito:

✔ Se ha cambiado la contraseña correctamente

Ahora es cuando el investigador puede realizar su labor: encolar un ataque cripto-analítico al supuesto de la factorización. Primero, debe seleccionar el supuesto a resolver:

Factorización
Logaritmo Discreto
Logaritmo Discreto sobre Curvas Elipticas
Colisiones de hash
Preimagenes hash

Al seleccionar “Factorización” se le pedirán los parámetros necesarios, como el número a factorizar y la cantidad de cores o núcleos a ocupar, dado que se resuelve en un ambiente distribuido:

Número a factorizar

239811859939591

Cores (Max. 4)

2

Submit

Luego de colocar los parámetros, el sistema le dirá que el trabajo está en proceso:

✔ Trabajo en proceso; dirijase al estado de tareas para ver su progreso

Como le dice el sistema, puede ver inmediatamente el tiempo que lleva o tomó la tarea, si ha terminado o no, y el resultado si es que está disponible:

Número	Estado	Input	Tiempo transcurrido	Resultado
0	Completo	Algoritmo: Factorización Número a factorizar: 239811859939591 Cores: 2	10 Segundo(s)	15485857·15485863

Ahora el investigador necesita resolver la factorización del mismo número, pero sobre más cores, para ir midiendo el tiempo que toma cada cálculo en función de los cores:

Número a factorizar

2398118599395912398

Cores (Max. 4)

4

Submit

Sin embargo, al ver el estado de las tareas, se da cuenta de que el número introducido era incorrecto y demasiado grande:

1	En proceso	Algoritmo: Factorización Número a factorizar: 23981185993959123981185993959123981185 99395912398118599395912398118599395912 39811859939591239811859939591239811859 939591239811859939591239811859939591 Cores: 4	53 Segundo(s)		Cancelar

Por lo cual, y dado que es posible, decide cancelar la tarea, para lo cual el sistema le dice que se logró cancelar correctamente:

✔ Se ha cancelado con éxito la tarea con id 208

Número	Estado	Input	Tiempo transcurrido	Resultado
0	Completo	Algoritmo: Factorización Número a factorizar: 239811859939591 Cores: 2	10 Segundo(s)	15485857·15485863
1	Cancelado	Algoritmo: Factorización Número a factorizar: 239811859939591239811859939591239811859 939591239811859939591239811859939591239 811859939591239811859939591239811859939 591239811859939591239811859939591 Cores: 4	4 Minuto(s) 47 Segundo(s)	Cancelado por el usuario

Luego de cancelarlo, encola nuevamente la factorización del número introducido anteriormente con cuatro cores. Sin embargo, al revisar el estado de tareas, ésta no parece avanzar debido a que se encuentra “Esperando Recursos”, es decir, espera que otras tareas terminen primero para que esta tarea tenga disponibles los cores que pidió:

2	Esperando Recursos	Algoritmo: Factorización Número a factorizar: 239811859939591 Cores: 4	0 Segundo(s)	

Es indefinido cuánto se quedará en este estado, pues depende netamente de las tareas que estan realizando el resto de los usuarios. Por suerte para el investigador, luego

de un momento, y al actualizar la página, se da cuenta que la tarea ya se encuentra completa:

2	Completo	Algoritmo: Factorización Número a factorizar: 239811859939591 Cores: 4	5 Segundo(s)	15485857·15485863

El resultado en tiempo de la ejecución, es como el investigador lo esperaba, se demoró la mitad de lo que se demoró el primero, dado que había el doble de cores en uso.

Finalmente, el investigador recuerda que tiene un trabajo relacionado con RSA y se pregunta si Cryptfront puede ayudarlo a esto, por lo que entra al servicio de "Subir tarea personalizada" y ve lo siguiente:

Archivo Tarea

Seleccionar archivo Ningún archivo seleccionado

Subir

O puede escribir su código acá:

```
override def descripcion = "RSA con n=3337 y e=79"

def run = {
    import scala.BigInt
    import supuestos._
    val n = BigInt("3337")
    val e = BigInt(79)

    val fact = Factorizar(n, 2).run //Puede ser Factorizar, DL, DLEC, EncontrarColision o Encontrar Preimagen

    val pattern = "([0-9]+)".r
    val seq = (for(m <- pattern.findAllMatchIn(fact)) yield m.group(1)).toList
    val p = BigInt(seq(0))
    val q = BigInt(seq(1))
    e.modInverse((p-1)*(q-1)).toString
}
```

Ejecutar

Con lo cual, decide probar el código en Scala puesto en el editor que justamente calcula la clave privada de RSA, sabiendo factorizar. Luego, el investigador revisa el estado de las tareas, encontrando la clave privada para la instancia de RSA del ejemplo:

3	Completo	RSA con n=3337 y e=79	1 Segundo(s)	1019

Con esto, el investigador se da cuenta de que podría realizar el trabajo recordado usando, sin problemas, la herramienta Cryptfront. Claro está, tendría que aprender algo de Scala antes de poder realizar su propio ataque criptoanalítico personalizado.

Capítulo 6

Conclusiones

En este trabajo de título se trabajó en el desarrollo de un sistema, que permite a los investigadores dedicados principalmente a la criptografía y la seguridad, ejecutar algoritmos criptoanalíticos implementados en una plataforma distribuida externa, eliminando el tiempo requerido para implementar o adaptar un algoritmo, y el tiempo requerido para encontrar un hardware adecuado, permitiendo al investigador optimizar el tiempo requerido para realizar ciertos trabajos. Para realizar esto, el trabajo se dividió en dos secciones: desarrollo de una plataforma web y configuración de un cluster en AWS.

Para el desarrollo de la plataforma web, se uso el framework Play! y el lenguaje Scala. En la plataforma se implementó lo siguiente: un login que permitiese autentificar a los usuarios, y asi reducir la entrada de usuarios malintencionados, al ser un sistema delicado; un sistema de creación de usuarios, en donde el administrador se encarga de crearlos y así no cualquier persona pueda registrarse; un sistema de resolución de supuestos que permite a un usuario ejecutar algoritmos criptoanalíticos incluidos en el sistema, el cual se conecta con el cluster para realizar tal labor; un mecanismo de extensión de supuestos que, de la mano del sistema de resolución de supuestos, permite a un desarrollador agregar un nuevo supuesto al sistema de forma sencilla y extensible, sin conocer mayores detalles del framework; un mecanismo para que un usuario pudiera realizar sus propios algoritmos criptoanalíticos basados en los ya existentes en el sistema, cuidando las clases que se pueden usar; y, finalmente un sistema que permite a los usuarios ver el estado de los algoritmos que se están ejecutando, paramétros, tiempo transcurrido y el resultado si esta disponible.

Con respecto a la configuración del cluster, además de instalar todas las librerías necesarias, se implementaron algoritmos distribuidos para resolver algunos de los supuestos más conocidos: encontrar la factorización de un número; encontrar el logaritmo discreto de un elemento cualquiera del grupo de los enteros módulo p con p primo, bajo la multiplicación y del grupo de las curvas elípticas sobre $\mathbb{F}_p$; encontrar una colisión en una función de hash; y encontrar una preimagen de una función de hash. Debido a la simplicidad, se implementaron algoritmos de tipo fuerza bruta con el uso de MPI para realizarlo distribuido.

Si bien el sistema descrito anteriormente, cumple la mayoría de los objetivos impuestos inicialmente, como lo es ejecutar un algoritmo criptoanalítico rapidamente, de manera

relativamente intuitiva, y siendo extensible para agregar otros algoritmos o ataques, hay ciertos puntos a considerar. Con respecto a la accesibilidad del sistema, éste no es de un acceso completamente sencillo, pues hay que ubicar y contactar a un encargado o administrador, explicarle la situación y esperar que cree una nueva cuenta para el usuario, lo que de cierta forma hace que el acceso inicial al sistema sea más lento de lo esperado. Sin embargo, luego de analizarlo con más detalle, este es un trade-off que debe existir, debido a la delicada naturaleza del sistema desarrollado. En segundo lugar, si bien es posible para un usuario realizar un algoritmo criptoanalítico personalizado, éste tiene que ser escrito en Scala, lo cual es una limitante para la persona que quiere usar esta sección del sistema, pues tiene que aprender algo de Scala antes de crear el nuevo ataque.

6.1. Trabajo futuro

Mejorar el diseño de la página web

El diseño actual de la página web, en lo que refiere a interfaz, es demasiado simple para los estándares actuales de la web. Por lo cual se propone re-estructurar los elementos de ésta, crear un menú desplegable con los servicios actuales, una mejor ubicación para las opciones de administrar al usuario (cambiar contraseña, logout, etc..). Todo esto puede ir acompañado por un rediseño de la página, ya sea con un diseño manual o con el uso de algún template CSS disponible en la web.

Realizar pruebas de usuario

Actualmente se encuentran desarrolladas las funcionalidades principales que permiten resolver el problema planteado. Sin embargo, no se han realizado suficientes pruebas con usuarios. Las pruebas con usuarios son importantes pues permiten validar la solución desde un punto de vista del usuario, quienes son, al final, los que usarán el sistema desarrollado.

Además de lo dicho anteriormente, las pruebas con usuarios permiten mejorar en gran forma la interfaz de la página web, por lo que se podría usar para mejorar el diseño de la página web.

Implementar o adaptar otros algoritmos en el cluster

En el cluster se encuentran implementados los algoritmos más básicos para la resolución de ciertos supuestos. Esto debido al tiempo disponible que este trabajo disponía, por lo cual se decidió poner prioridad a otras cosas que eran importantes para la aplicación. Sin embargo es importante también que se incluyan otros algoritmos más sofisticados

y más rápidos, que le permitan dar una utilidad real al sistema, por ejemplo, el Number Field Sieve distribuido de [43], ya que los criptógrafos trabajan con secuencias o números de una gran cantidad de bits, que con un algoritmo ineficiente, aunque distribuido, pueda tardar demasiado tiempo para que sea útil.

Por otro lado, es útil tener todo tipo de implementaciones, pues se pueden integrar todos los algoritmos, ya que algunos algoritmos específicos son más rápidos que otros para cierto rango de entrada.

Mostrar al usuario un estado del progreso de las tareas

Un estado del progreso de una tarea sería útil para un usuario de este sistema, pues le daría información sobre cuánto aproximadamente falta para que termine la tarea, lo cual es novedoso pues los algoritmos criptoanalíticos no tienden a mostrar el progreso que llevan. Un ejemplo de esto, con el algoritmo de factorización de fuerza bruta, es que cada vez que recorre en un 1 % el espacio de posibles factores, avise de alguna forma al servidor web, y este lo actualiza en la fila de la base de datos correspondientes, y así cuando el usuario vea el estado de las tareas, vea el progreso reflejado en la tabla.

Para implementar esto, se podría hacer que los algoritmos implementados en MPI, ejecuten una función que recibe un progreso dependiente del algoritmo y lo mande a la base de datos del servidor web para actualizar el estado, resolviendo problemas de concurrencia de los distintos procesos.

Permitir al usuario agregar nuevas resoluciones de problemas computacionales al sistema

Con la solución actual, el usuario tiene que escribir o subir cada vez el script correspondiente a una resolución de un supuesto personalizado con parámetros distintos, lo cual lo hace poco flexible. Una solución a esto sería hacer que el usuario pueda agregar estos nuevos supuestos al sistema, implementando su resolución. Con esta medida, otros usuarios también podrían usar estos supuestos implementados y así evitar que todos los usuarios tengan que reimplementar una y otra vez una resolución de RSA, por ejemplo.

Permitir parar y reanudar una tarea

El sistema desarrollado permite cancelar una tarea, pero no pararla y reanudarla. Lo último permite hacer el sistema aún más flexible y orientado al usuario. Por ejemplo, el usuario ha ejecutado un algoritmo que demora una semana con los parámetros introducidos, sin embargo, al día después se da cuenta que necesita de forma urgente el resultado del mismo algoritmo, pero con otros parámetros, el cual demora un día, pero no hay recursos en el cluster. Con la solución actual no podría realizar aquella labor; tendría que

cancelar la tarea existente e iniciar una nueva, o esperar la semana que tarda la tarea anterior. Si se pudiese parar y reanudar una tarea, el usuario podría resolver su problema fácilmente.

Para implementar esto, habría que guardar el estado del algoritmo corriendo actualmente al disco para pararlo y recuperar este estado del disco al reanudarlo. Como todos los algoritmos de todos los supuestos tienen variables distintas y funcionan de distinta forma, es necesario que cuando se quiera agregar un nuevo algoritmo, se implemente una función genérica que guarde el estado a disco y otra que recupere el estado del disco, y de esta forma, no mezclar código de la resolución del supuesto con el código que guarda y recupera el estado.

Permitir a un administrador agregar o quitar instancias del cluster en AWS

Actualmente para que el sistema detecte que se ha agregado una nueva instancia al cluster de AWS, hay que reiniciar el servidor web, por lo cual es poco práctico, pues al reiniciar el servidor web, se pierden las ejecuciones de las tareas, así que habría que esperar que terminen todas las tareas antes de reiniciar el servidor web.

Por esto, se propone que el sistema soporte el agregar o quitar instancias del cluster, dentro de la misma página web. Para esto habría que usar la API de AWS para agregar o iniciar instancias, y para parar las instancias. Con esto se podría actualizar de manera automática en el sistema web, la cantidad y la dirección de los nuevos nodos del cluster, cuidando de parar todas las tareas que esten usando nodos del cluster que van a ser detenidos para evitar estados inconsistentes.

Evaluar uso de Wisecracker

En la sección de Soluciones Existentes se mencionó la existencia de Wisecracker, una herramienta que abstrae el uso de MPI (y de OpenCL). Esta herramienta está enfocada en facilitar el trabajo realizado por los investigadores en sus implementaciones de algoritmos criptoanalíticos sobre los cuales trabajan. El sistema implementado en este trabajo, se basa principalmente en que los algoritmos criptoanalíticos deben ser adaptados y no re-implementados, debido a la complejidad que representa el re-implementar un algoritmo especifico. Es por esto que Wisecracker podría ser usado para adaptar un algoritmo secuencial a un algoritmo distribuido e, incluso, podría ser usado si es que se llegase a implementar un algoritmo criptoanalítico desde cero. Sin embargo, como Wisecracker fue encontrado en las etapas finales de este trabajo, se debe evaluar su uso y si realmente facilita el realizar implementaciones distribuidas sobre MPI.

Bibliografía

[1] Volunteer Computing. http://en.wikipedia.org/wiki/Volunteer_computing

[2] Paul C. van Oorschot and Michael J. Wiener. ”Parallel Collision Search with Application to Hash Functions and Discrete Logarithms”. http://www.certainkey.com/dnet/acmccs94.pdf

[3] P. Flajolet and A.M. Odlyzko, “Random Mapping Statistics”, Lecture Notes in Computer Science 434: Advances in Cryptology - Eurocrypt ’89 Proceedings, Springer-Verlag, pp. 329-354.

[4] D.E. Knuth, “The Art of Computer Programming, vol. 2: Seminumerical Algorithms”, 2nd edition, Addison-Wesley, 1981.

[5] Victor Shoup. “A Computational Introduction to Number Theory and Algebra”, Second Edition. http://www.shoup.net/ntb/ntb-v2.pdf.

[6] I. N. Herstein. “Abstract Algebra”, Third Edition.

[7] D. Shanks. “Class number, a theory of factorization and genera. In Proc. Symp. Pure Math. 20”, pages 415—440. AMS, Providence, R.I., 1971.

[8] Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman. “An Introduction to Mathematical Cryptography.”

[9] Tim Güneysul, Gerd Pfeiffer, Christof Paar, Manfred Schimmler. “Three Years of Evolution: Cryptoanalysis with COPACOBANA”. http://www.emsec.rub.de/media/crypto/veroeffentlichungen/2010/09/07/sharcss09_copacobana.pdf

[10] FIPS Publication 46-3. http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

[11] EFF “EFF DES Cracker”. https://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_descracker_pressrel.html

[12] DES Challenges. http://en.wikipedia.org/wiki/DES_Challenges.

[13] distributed.net. http://www.distributed.net/Main_Page

[14] RIVYERA S6-LX150. http://www.sciengines.com/products/computers-and-clusters/rivyera-s6-lx150.html

[15] GIMPS. http://mersenne.org/

[16] BOINC. http://boinc.berkeley.edu/

[17] Ronald L. Rivest. “The RC5 Encryption Algorithm”. http://people.csail.mit.edu/rivest/Rivest-rc5rev.pdf

[18] NFS@home. http://escatter11.fullerton.edu/nfs/

[19] N. Naadimuthu. “A Java Framework for Cryptanalysis using GA (JFCGA)”. http://www.ijmttjournal.org/Volume-1/Issue-1/ijmttjournal-v1i1p3.pdf

[20] Antonio Marcos de Oliveira Candia. “Flexible Cryptanalysis in Java”. http://www.ijofcs.org/V01N1-P06 %20- %20Flexible %20Cryptanalysis %20in %20Java.pdf

[21] Vikas N Kumar. “A high performance distributed cryptanalysis framework”. http://selectiveintellect.com/wisecracker_whitepaper.pdf

[22] The OpenCL Specification. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[23] MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[24] Slick, Scala Language Integrated Connection Kit. http://slick.typesafe.com/

[25] ”A Future-Adaptable Password Scheme”. https://www.usenix.org/legacy/events/usenix99/provos/provos_html/node1.html

[26] Akka. http://akka.io/

[27] Play 2.2 Documentation. http://www.playframework.com/documentation/2.2.x/Home

[28] NTL. http://www.shoup.net/ntl/

[29] GMP, GNU Multiple Precision Arithmetic Library. https://gmplib.org

[30] MIRACL. http://www.certivox.com/miracl/

[31] CryptoPP. http://www.cryptopp.com/

[32] R. Schoof. “Elliptic curves over finite fields and the computation of square roots mod p”. http://www.mat.uniroma2.it/~schoof/ctpts.pdf

[33] Douglas R. Stinson. “Cryptography: Theory and Practice”, Second Edition.

[34] yafu. http://sourceforge.net/projects/yafu/?source=dlp.

[35] Carl Pomerance. “A Tale of Two Sieves”.

[36] Kleinjung, Thorsten (October 2006). “On polynomial selection for the general number field sieve”.

[37] Elliptic Curve Discreet Logarithm Problem - Pollard’s rho-method.
https://code.google.com/p/ecdlp-pollard/source/browse/

[38] Implementation of the Pohlig-Hellman Algorithm for the Discrete Logarithm Problem. http://alum.wpi.edu/~sorrodp/crypto/report.html

[39] Richard P. Brent. “Parallel Algorithms for Integer Factorisation”.

[40] PollardRho.java. http://introcs.cs.princeton.edu/java/78crypto/PollardRho.java.html

[41] Factorization Source Code. http://www.frenchfries.net/paul/factoring/source.html

[42] Quadratic-Sieve. https://github.com/martani/Quadratic-Sieve

[43] Cado-NFS. http://cado-nfs.gforge.inria.fr/#feat

[44] Baby-step giant-step. https://github.com/viralpoetry/Baby-step-giant-step

[45] Amazon EC2. http://aws.amazon.com/ec2

[46] Amazon Web Services Documentation. http://aws.amazon.com/documentation/

[47] Amazon EC2 Instance Types. http://aws.amazon.com/ec2/instance-types/

[48] MPICH-2. http://www.mpich.org.

[49] Scala documentation. http://docs.scala-lang.org/.