Universidad de Buenos Aires
Facultades de Ciencias Económicas,
Ciencias Exactas y Naturales e
Ingeniería

Maestría en Seguridad Informática

Tesis

Hash Collider
Desarrollo de un sistema distribuido para generar colisiones en funciones de *hashing* con documentos de contenido arbitrario y semánticamente correcto

Autor: Juan Alejandro Knight
Director de Tesis: Hugo Scolnik
Co-director: Pedro Hecht

Fecha de Presentación: Agosto 2012
Cohorte: 2011

Declaración Jurada de origen de los contenidos

Por medio de la presente, el autor manifiesta conocer y aceptar el Reglamento de Tesis vigente y que se hace responsable que la totalidad de los contenidos del presente documento son originales y de su creación exclusiva, o bien pertenecen a terceros u otras fuentes, que han sido adecuadamente referenciados y cuya inclusión no infringe la legislación Nacional e Internacional de Propiedad Intelectual

FIRMADO

Juan Alejandro Knight

DNI 30.449.419

Hash Collider – Desarrollo de un sistema distribuido para generar colisiones en funciones de *hashing* con documentos de contenido arbitrario y semánticamente correcto

Resumen

Las funciones de *hashing* son funciones que permiten obtener un identificador (*hash*) de un documento según su contenido. Tienen cuantiosísimas aplicaciones en el campo de la seguridad informática ya que permiten detectar de forma rápida y concluyente las más leves modificaciones en un documento.

Por razones prácticas de diseño, un *hash* puede estar vinculado a más de un documento, por consiguiente generando una colisión. Si un atacante consigue una colisión y reemplaza el documento original con el fraguado, los controles de integridad serán salteados. Si el documento fraguado tuviese como contenido lo que el atacante desea, entonces el fraude será efectivo, lo cual trae serias implicancias en el resguardo de la información.

Este trabajo ofrece una herramienta innovadora llamada HashCollider la cual representa el aporte del autor. Mediante el uso de la misma, se podrán generar estas colisiones con documentos fraguados inteligentemente creados para efectuar estos tipos de fraudes. El sistema es distribuido y colaborativo, permitiendo que un ejército de equipos pueda trabajar de forma paralela, coordinada, simple y eficiente vía Internet. Si bien la herramienta es una prueba de concepto, las consecuencias de hallar estas colisiones serían revolucionarias.

Palabras Clave

Seguridad Informática – Función de *hashing* – Colisión de *hash*

Contenido

Declaración Jurada de origen de los contenidos 2
Resumen 3
Palabras Clave 3
Nómina de abreviaturas 7
Cuerpo Introductorio 8
Área temática 8
Términos Utilizados 8
Introducción 10
Objetivo 12
Alcance 12
Hipótesis 13
Cuerpo Principal 14
1. Conceptos teóricos 14
1.1. Funciones de *hashing* 14
1.2. Propiedades de las funciones de *hashing* 14
1.3. Definición matemática de las funciones de *hashing* 15
1.4. Usos de las funciones de *hashing* 16
1.5. Seguridad de funciones de *hashing* 17
1.6. Firmas digitales 20
1.7. Colisión de *hash* con documento arbitrario 22
1.8. Estado del arte 24
2. Generación de colisiones con HashCollider 27
2.1. Desafíos 27
2.2. Tipos de desafíos 28
2.3. Fragmentos 28
2.4. Diseño del sistema 31
2.5. Plataforma 32
2.6. Algoritmos de *hashing* utilizados 33
2.7. Motor de fragmentación 33
2.8. Resolución de fragmentos 35
2.9. Estrategias en el armado de *padding* 36
2.10. *Buffer Caché* en el motor de fragmentación 37
2.11. Desplazamiento de fragmentos en clientes 38

2.12. Pruebas integrales 40
2.13. Publicación web del avance 42
Métricas de desempeño 43
Entorno de ejecución 43
Caudal de documentos procesados 43
Escalabilidad 43
Conclusiones 46
Pruebas de la hipótesis 46
Mejoras futuras y posibles líneas de profundización 48
Escalabilidad y efectividad 49
Anexo A: *Web Services* 50
Definición 50
Propósito 50
Anexo B: Requerimientos del Sistema 51
Anexo C: Diseño del Sistema 53
Diagrama de Despliegue 53
Diagrama de Procesos 53
Diagrama de Componentes 54
Diagrama de Actividad 56
Diagrama de Uso 56
Anexo D: Extensión de funcionalidad 58
Anexo E: Protocolo de comunicación 60
Métodos publicados 60
Protocolo de comunicación 61
Anexo F: Acceso *web* a HashCollider 63
Página de inicio 63
Avance en la resolución de los desafíos 64
Estrategias de resolución 65
Descarga del *software* cliente 66
Resumen de aspectos teóricos 66
Notificaciones vía email 67
Acceso al panel de administración 68
Administración de desafíos 68
Crear un nuevo desafío 70

Editar un desafío existente ... 71
Fuentes ... 72
Bibliografía ... 73
Bibliografía General ... 75
Índice de Ilustraciones y Tablas ... 76

Nómina de abreviaturas

- CA (*Certification Authority*): Es una organización a la que dos partes deben conocer y confiar para establecer el origen de las partes y un mecanismo de autenticación de mensajes.
- HMAC (*Keyed-Hashing for Message Authentication*): Es un mecanismo criptográfico que permite autenticar un mensaje y verificar su origen.
- MD5 (*Message Digest* 5): Es una función de *hashing* creada en 1991 y ampliamente utilizada por su velocidad.
- SHA1 (*Secure Hash Algorithm*): Es una función de *hashing* creada en 1993 para reemplazar a MD5 y en ese entonces era considerada la opción más segura.
- SQL (*Structured Query Language*): Es un lenguaje declarativo para ejecutar consultas en una base de datos.
- XML (*Extensible Markup Language*): Es un formato de texto flexible y escalable para la estructuración e intercambio de datos de forma sencilla de procesar. Se utiliza una jerarquía de etiquetas y atributos para modelar los datos. Debido a su simpleza es ampliamente utilizado para la transmisión de datos en medios digitales.
- X509: Es un estándar para un certificado digital emitido por una autoridad certificante y utiliza criptografía de clave pública para asegurar el origen y contenido de un mensaje.

Cuerpo Introductorio

Área temática

El campo de trabajo está principalmente situado en la seguridad informática. Se abordarán temas más específicos como las funciones de *hashing* y en menor grado las firmas digitales. El foco estará colocado en la seguridad que ofrecen los esquemas de verificación de integridad de documentos que utilizan funciones de *hashing*. Particularmente, se detallará sobre la generación de colisiones de *hash* con documentos con contenido arbitrario y semánticamente correcto.

Términos Utilizados

Los términos utilizados se resumen a continuación:

- Seguridad Informática: Área orientada al resguardo de la información y la infraestructura computacional que lo sostiene. La seguridad informática está compuesta por tres pilares: confidencialidad, integridad, disponibilidad. Básicamente, la información debe ser accesible únicamente por el personal autorizado y no debe ser modificado sin tener dicha potestad. La seguridad informática no se puede asegurar implementando únicamente algoritmos criptográficos, protocolos y ni siquiera adquiriendo tecnología de punta. Por el contrario, se deben instaurar procedimientos y todos los involucrados deberán cumplirlos.
- Criptografía: Es el estudio de métodos y algoritmos matemáticos para poder transformar un mensaje legible en algo ilegible que pueda ser descifrado únicamente por las personas autorizadas. La criptografía está íntegramente vinculada con ciertos aspectos de la seguridad informática como la confidencialidad, integridad de datos, autenticación de las partes involucradas e identificación del origen de los datos. Si bien se la utilizó varios siglos antes de la creación de la primera computadora, el poder de cómputo en sistemas digitales propulsó en gran manera su complejidad y variedad de ataques posibles [1].

- Criptoanálisis: Es el estudio riguroso de las debilidades y vulnerabilidades de distintos esquemas criptográficos para crear ataques de manera tal de quebrar su seguridad [1].
- Encripción: Acción de transformar un mensaje legible generado por cualquier persona (o máquina con dicho atributo) a uno ilegible el cual sólo pueda ser interpretado por las partes autorizadas [1].
- Criptosistema: Se refiere a un conjunto de herramientas criptográficas que proveen en algún sentido seguridad informática. Sus aseveraciones más frecuentes son utilizadas para identificar sistemas que provean confidencialidad al encriptar la información [1].
- Función de *hashing* criptográfica: Es una función matemática que toma como entrada a un documento, cualquiera sea su tamaño, y produce una salida de tamaño fijo denominado *hash* [1]. La función es determinística y repetible, es decir, ante una misma entrada, la salida siempre será la misma. La idea principal de las funciones de *hashing* criptográficas es que la distribución del espacio de *hashes* posible debe ser lo más cercana a la distribución uniforme. Además, si sólo se altera un bit en el documento de entrada, la salida no debe tener ninguna correlación con el *hash* original, lo que permite implementar controles para detectar modificaciones de un documento.
- Colisión: Las funciones de *hash* tienen una relación “N a 1”, es decir, existe más de un documento asociado al mismo *hash* [1]. Al tener el *hash* un tamaño fijo su espacio de valores posibles es finito por lo que existirán distintas entradas que tengan asociado un mismo *hash*. Este fenómeno se denomina colisión.
- Firma digital: Es un proceso análogo a la firma hológrafa realizada a mano con una pluma o bolígrafo. Para efectuar una firma de forma digital, se utilizan funciones matemáticas y criptográficas. La misma podrá ser verificada de forma pública, es decir, cualquier persona puede verificar si un documento fue firmado por quien dice ser el firmante. Otra característica en este esquema es el no repudio: una vez firmado, el firmante no puede negar haber realizado la firma. Más importante aún, cualquier modificación posterior a la firma será detectada resultando en una firma inválida [1].

- Seguridad computacional: Es un indicador de cuán seguro es un criptosistema según el tiempo y poder de cómputo necesarios para comprometer su seguridad. Si un criptosistema es computacionalmente seguro, se requerirá de un gran poder de cómputo dedicado (el cual debe ser difícil de conseguir) por un tiempo tan largo, que el ataque deja de ser práctico. La practicidad de un ataque se ve severamente deteriorada por el costo asociado de quebrar el criptosistema y el tiempo que tardará dicho ataque [1]. En muchos de los casos, la información resguardada por el criptosistema puede tener un costo no comparable con el costo del ataque, o bien ya no tendrá valor una vez consumado el ataque.
- Semántica de documento: La semántica de un documento se particulariza por una serie de reglas de formato que se deben cumplir. Las reglas de formato dependerán según corresponda a un tipo de documento específico. Tomando como ejemplo a un archivo XML [2], el contenido debe estar bien formado, es decir, cada etiqueta abierta debería estar cerrada acordemente. Además, debe haber un nodo principal y cada atributo debe estar encerrado entre comillas.
- Documento semánticamente correcto: Es un documento que cumple con las especificaciones semánticas según el tipo de documento al que pertenezca.

Introducción

En el mundo actual, las necesidades del negocio dependen en gran manera de sistemas de información, comunicación por medios digitales, envío de documentación y cumplir con flujos de trabajo. Mucha de la información y documentos utilizados son digitales, dejando de lado el soporte en papel. La seguridad de dicha información debe contemplar la confidencialidad, integridad, disponibilidad y el no repudio. Los documentos digitales pueden ser fácilmente editados y copiados casi sin dejar rastros, lo cual afecta directamente la integridad y el no repudio de los mismos.

¿Cómo se puede resguardar la información digital de alteraciones no autorizadas? La respuesta es el uso de funciones de *hashing*. Cabe destacar que en este trabajo siempre que se mencione una función de *hashing*, se

está haciendo referencia a una función de *hashing* criptográfica. Las funciones de *hashing* permiten calcular un *hash* (identificador binario) a partir de un documento. La idea principal de una función de *hashing* es que al cambiar tan sólo una mínima modificación en el documento, el *hash* calculado debe ser distinto e independiente del *hash* original. Es decir, la función de *hash* permite detectar modificaciones en el documento y es una herramienta crítica para asegurar la integridad de la información de posibles fraudes, modificaciones no autorizadas y detectar la pérdida de integridad.

Las firmas digitales permiten verificar la fuente de un documento firmado, detectar modificaciones en el documento luego de ser firmado y que el firmante no pueda rechazar el hecho de que efectivamente haya firmado el documento. Además, las firmas digitales pueden ser fácilmente integradas en sistemas automatizados. Fueron diseñadas utilizando diversos algoritmos criptográficos, incluyendo criptosistemas y las ya mencionadas funciones de *hashing*.

Las funciones de *hashing* tienen por diseño y practicidad, un espacio de *hashes* finito a causa del tamaño fijo del *hash*. Esta característica conlleva a la existencia de colisiones que ocurren al existir varios documentos posibles asignados al mismo *hash*. La colisión en este caso no permite identificar la alteración de los datos ya que ambos documentos tienen el mismo *hash*. Una colisión podría ser utilizada por un atacante para reemplazar documentos críticos.

Una de las medidas de seguridad de una función de *hashing* es complejizar la búsqueda de colisiones. Cuanta menos probabilidad haya de encontrar una colisión, mayor será el procesamiento necesario para hallarla. Cuanto más difícil sea hallar una colisión, mayor seguridad computacional tendrá la función de *hashing*, ya que se requerirá de mucho tiempo para efectuar el ataque, anulando toda practicidad a estos tipos de ataques.

Básicamente, una firma digital está compuesta por un *hash* encriptado. Si existe una colisión, entonces ambos documentos tendrán el mismo *hash* y por ende la misma firma digital. El documento fraguado que entra en colisión estará firmado digitalmente sin el consentimiento del firmante y la adulteración será indetectable.

¿Qué sucedería si se pudiesen obtener documentos fraguados que logren una colisión? ¿Qué consecuencias habría en la seguridad de la información? ¿Cómo se podrían obtener estos documentos fraguados? ¿Se pueden obtener documentos que colisionen y que contengan la información que uno desee? Este trabajo intentará contestar todas estas preguntas.

Objetivo

La meta de este trabajo es proveer un sistema que permita hallar documentos fraguados que generen una colisión. La herramienta es una prueba de concepto y representa el aporte del autor. El documento fraguado tendrá un contenido arbitrario e intencionalmente escogido y será semánticamente correcto.

Las implicancias de hallar colisiones de *hash* pueden facilitar ataques de manipulación de datos, fraude, robo de identidad, entre muchas otras. Al hallar una colisión, se podrá realizar una suplantación del documento original burlando controles de verificación de integridad. Además, si se hubiese utilizado un esquema de firmas digitales, se podrán firmar documentos con el contenido deseado y sin el consentimiento del firmante.

Alcance

El presente trabajo tiene como fin realizar un aporte práctico a los ataques de colisiones de funciones de *hashing*. Si bien esta herramienta conforma una prueba de concepto, la idea principal es crear un sistema distribuido completamente innovador que permita generar colisiones de *hash* con un documento de contenido escogido y semánticamente correcto.

El sistema utilizará un *web service* para coordinar la resolución distribuida de las colisiones. Todo el sistema será desarrollado en .Net [7] orientado a una plataforma Windows. A futuro, se podría desarrollar clientes para otras plataformas, tal como Linux, para ampliar el espectro de posibles clientes y aumentar el caudal de procesamiento distribuido. Para ver más detalles sobre los *web services*, se recomienda leer primero el Anexo A.

Se construirá un sitio *web* que permita ver el avance en línea de la resolución de los desafíos.

Hipótesis

HashCollider es una herramienta que facilitará la generación de colisiones de *hash* en base a un documento de contenido arbitrario y escogido, respetando las reglas semánticas según el tipo de documento.

Cuerpo Principal

1. Conceptos teóricos

Antes de poder ahondar en los aspectos más técnicos, es necesario explayar los conceptos claves de los cuales se nutre HashCollider. Para poder comprender cómo funciona el sistema, por qué fue construido y con qué fin, se requiere tener un conocimiento básico sobre algunas cuestiones de la seguridad informática. Se presume que el lector está familiarizado con el tema, sin embargo, la introducción servirá para exponer las ideas fundamentales sobre el cual se basa este trabajo y para proveer una plataforma para alinear y relacionar los conceptos aquí expuestos.

La generación de colisiones de *hash* tiene su problemática intrínseca, la cual debe ser desmenuzada para una mejor comprensión de la herramienta. A continuación se repasarán los temas que atañen a las funciones de *hashing*, por qué se utilizan, dónde se utilizan y por qué existen las colisiones.

1.1. Funciones de *hashing*

Las funciones (o algoritmos) de *hashing* son el resultado de aplicar una serie de cálculos y operaciones a un documento de entrada. El documento de entrada es de carácter digital y está compuesto por una cadena binaria. El resultado de tal algoritmo se llama *hash*, el cual representa un identificador de dicho documento determinado según su contenido [1].

Ilustración 1 - Función de *Hashing*

1.2. Propiedades de las funciones de *hashing*

Las funciones de *hashing* son funciones especiales que deben cumplir con varias propiedades [1]:

- Las funciones de *hashing* no son estocásticas sino determinísticas. No dependen del azar o de alguna información desconocida. El documento es el único dato de entrada para una función de *hashing*.
- El tamaño del documento de entrada es incierto por lo que la función de *hashing* debe tolerar documentos de cualquier tamaño.
- No hay restricciones en el contenido del documento de entrada sin importar su tipo (texto o binario) ni formato (texto, imagen, video, audio, entre muchas otros).
- El tamaño del hash es fijo y conocido según la implementación de la función y se mide en cantidad de *bits*. Por ejemplo, dos funciones de *hashing* muy conocidas son SHA1 que genera *hashes* de 160 *bits* [3] y MD5 de 128 *bits* [4].
- Las funciones de *hashing* deben ser públicas o al menos conocidas por todas las partes involucradas.
- Las funciones de *hashing* son repetibles, por consiguiente, el *hash* de un documento deberá ser siempre el mismo sin importar cuántas veces se calcule.
- Si cualquier *bit* es modificado en el documento, el *hash* resultante debe ser totalmente distinto del original y no tener ninguna correlación.
- Los *hashes* deben ser rápidamente calculables. Para que las funciones de *hashing* sean prácticas de utilizar, los algoritmos deben ser veloces y ofrecer un tiempo de respuesta mínimo.
- El proceso inverso debe ser complejo y computacionalmente seguro. En otras palabras, al conocer un *hash* debe ser prácticamente imposible obtener qué documento lo generó.
- El algoritmo debe ser resistente a distintos ataques criptoanalíticos.

1.3. Definición matemática de las funciones de *hashing*

La definición matemática de una función de *hashing* h es $h: D \rightarrow R$, donde D representa al dominio circunscripto a t *bits* y R el rango con un tamaño fijo de n *bits*. El dominio representa el espacio de todos los

documentos de entrada que pueden ser utilizados. El rango de la función *h* es el espacio de los posibles valores del *hash* resultante.

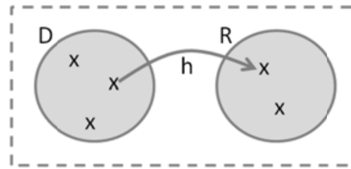


Ilustración 2 - Conjuntos de una función de *hashing*

Por razones prácticas se cumple que $|D| > |R|$, es decir, el espacio de documentos es mayor al espacio de *hashes*. La solución a una función donde $|D| \leq |R|$ es trivial ya que, en ese caso, habría más *hashes* disponibles que documentos permitiendo fácilmente la existencia de una función unívoca.

Considerando las longitudes de los documentos y *hashes* se puede afirmar $n < t$. Este tipo de funciones se denominan del tipo “muchos a uno”. Esta característica implica la existencia de colisiones dado que existen más documentos que *hashes*, por consiguiente, existirán subconjuntos de documentos que tengan el mismo *hash* asociado.

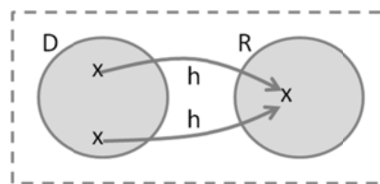


Ilustración 3 - Conjuntos en una colisión de *hash*

La cantidad total de colisiones se calcula dividiendo la cantidad de documentos 2^t por la cantidad de *hashes* 2^n, resultando en 2^{t-n} colisiones. La probabilidad de encontrar una colisión es 2^{-n}, lo cual no depende de *t* [1].

1.4. Usos de las funciones de *hashing*

Las funciones de *hashing* crean rápidamente identificadores de documentos, o también llamados imágenes. Uno de los usos más frecuentes de una función de *hashing* es el de detectar modificaciones en un documento. Cualquier alteración, por más mínima que sea, un algoritmo de *hashing* seguro debería dar un *hash* que sea totalmente independiente y sin correlación con el *hash* del documento original. Este mecanismo permite

detectar si un documento fue adulterado, asegurando la integridad del mensaje.

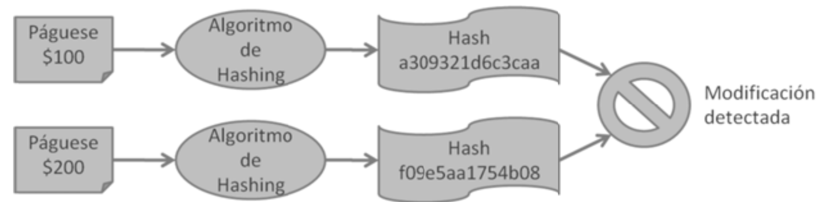


Ilustración 4 – Detección de documentos adulterados comparando los *hashes*

Debido a las características únicas de las funciones de hashing, éstas tienen diversos usos:

- Verificación de integridad de información al poder detectar mínimas alteraciones.
- Esquemas de firmas digitales.
- Autenticación del origen de mensajes mediante un HMAC (*Hashing Message Authentication Code*) [5]. Estas funciones utilizan una clave secreta como entrada adicional, la cual es conocida únicamente por las partes involucradas.
- Almacenamiento del *hash* resultante de las contraseñas de sistemas y así evitar persistirlas en texto claro.
- Identificación de virus al tener una base de datos de archivos maliciosos.
- Pericias en un análisis forense en la búsqueda de evidencia.
- Embeber de forma imperceptible evidencias de autoría de documentos digitales a ser verificados en caso de un litigio. Esta técnica se denomina *digital watermarking* [6].

1.5. Seguridad de funciones de *hashing*

Para que una función de hashing sea considerada criptográficamente segura, ésta debe cumplir con una serie de requisitos que aseguren la resistencia a ciertos tipos de ataques. El grado de resistencia mide la capacidad de la función de hashing para no ser vulnerada utilizando un ataque criptoanalítico [1].

- Resistencia de pre-imagen: Dado un documento desconocido y el *hash* conocido, es computacionalmente inviable hallar un documento que genere dicho *hash*. Este tipo de resistencia es la causa por la que las funciones de *hashing* son llamadas funciones de una vía. El cálculo del *hash* es una tarea sencilla y rápida, pero por otro lado el proceso de revertir el *hash* a su documento precursor es un proceso complejo y computacionalmente inviable.

Ilustración 5 – Resistencia de pre-imagen

A modo de ejemplo, un atacante ha logrado obtener la tabla de usuarios y contraseñas de un sistema *web* mediante el uso de SQL *Injection* [8]. Al indagar en el contenido de dicho descubrimiento, el atacante nota que las contraseñas no están almacenadas en texto claro y responden a un campo hexadecimal en la columna que se llama “SHA1password”. El atacante pudo obtener el hash de las contraseñas e intuye que el algoritmo de *hashing* utilizado es SHA1 [3]. En este escenario, el atacante conoce la función de *hashing* y el *hash* de las contraseñas. El objetivo claramente es determinar qué contraseña genera cada *hash* obtenido utilizando el algoritmo SHA1 [3].

Para que sea considerado una función de *hashing* segura, la labor de obtener dichas contraseñas debería ser ardua ya que no existe una forma sencilla de revertir el hash de una forma eficiente. La mejor opción del atacante es realizar un ataque de diccionario en el cual se utilice un listado de contraseñas frecuentes para intentar acelerar el proceso. Si el atacante no tiene suerte, deberá realizar una búsqueda de la contraseña por fuerza bruta.

- Segunda resistencia a la pre-imagen: Dado un documento conocido, es computacionalmente inviable hallar otro documento que genere una colisión. El concepto radica en que no debería haber una correlación

fuerte entre los *hashes* y el conjunto de documentos modificados en al menos un *bit*.

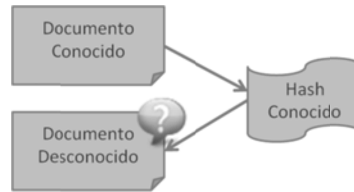


Ilustración 6 – Segunda resistencia a la pre-imagen

Un ejemplo de este tipo de resistencia se puede identificar en un servidor de descarga de archivos. Generalmente cuando una persona o institución publica un documento, también le adjunta el *hash* de dicho documento e informa con qué algoritmo se calculó. La idea principal de este procedimiento es que los usuarios puedan descargar dicho archivo, calcular el hash con el algoritmo especificado y comparar si los *hashes* son idénticos. Si los hashes son distintos se puede asegurar que el contenido del documento fue adulterado.

En el caso de que un atacante obtenga un vector de ataque con los suficientes privilegios para modificar los archivos publicados por un tercero, cualquier modificación debería generar un hash totalmente distinto al publicado. Los usuarios detectarán fácilmente la adulteración del archivo y muy posiblemente se le comunique al administrador del sitio. La labor del atacante de buscar un documento levemente modificado que colisione con el documento original debe ser computacionalmente segura.

- Resistencia de colisiones: Es computacionalmente inviable hallar dos documentos cualesquiera que generen una colisión. No se tiene ninguna información sobre ninguno de los dos documentos. Esto implica que no debe haber debilidades o atajos en el algoritmo que permitan fácilmente obtener documentos sinónimos que generen una colisión.

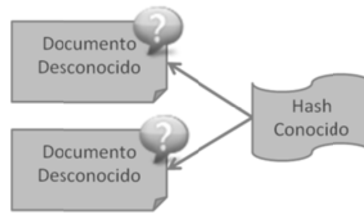


Ilustración 7 – Resistencia de colisiones

Por ejemplo, si se utilizara la función modular como algoritmo de hashing, esta función no cumpliría este requisito. Definiendo a la función como $h(d) = d \bmod q$ siendo q un número entero por el cual se aplica el módulo, se puede determinar fácilmente la familia de sinónimos que generarán colisión. Utilizando la propiedad matemática $q \bmod q = 0$, basta con sumar q al documento para obtener una colisión. Sin tener conocimiento de ningún documento, es posible generar colisiones rápidamente. La familia de sinónimos estará definida como $s(d) = d + kq$, donde k es un número entero cualquiera y el conjunto de posibles colisiones será $\{\ldots, d - 2q, d - q, d, d + q, d + 2q, \ldots\}$.

1.6. Firmas digitales

La firma digital es análoga a la firma manuscrita y sirven para constatar el origen y autenticidad del origen de la información. Las mismas dependen de un secreto, únicamente conocido por el firmante, y a su vez del contenido que se desea firmar. Cualquiera alteración en el secreto o en el contenido firmado será fácilmente detectada. El firmante podrá firmar digitalmente un documento para asegurar la integridad del contenido y constatar la identidad del responsable de dicha firma [1].

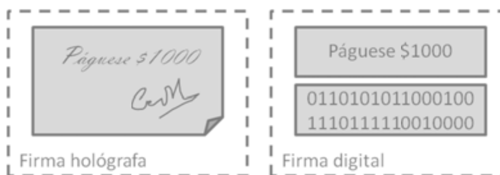


Ilustración 8 – Analogía entre firma hológrafa y digital

Las firmas digitales son esquemas que utilizan varios métodos criptográficos. Para poder verificar la integridad, se utilizan funciones de

hashing para generar un *hash* del documento a firmar, el cual luego será cifrado con una clave secreta conocida únicamente por el firmante [1].

Las firmas digitales deben ser verificables. Llegado el caso de una disputa entre las partes involucradas sobre quién firmó el documento o cuál era el contenido firmado, la resolución del conflicto debe ser clara, concisa y contundente. El litigio se puede ocasionar si el firmante repudia un documento firmado y asegura no haberlo firmado, o bien un demandante alega que el firmante supuestamente firmó un documento. La verificación se debe realizar sin exponer el secreto únicamente conocido por el firmante y requiere de un tercero que sea confiable.

Ilustración 9 – Firma digital

La firma digital permite asegurar el no repudio, una de las dimensiones que conforman la seguridad informática. Esta propiedad permite asociar la identidad del firmante a un mensaje firmado. Dado que la firma se genera mediante un secreto conocido únicamente por el firmante, se puede aseverar que la firma la realizó únicamente el firmante y de forma consciente. Una vez firmado el documento, el firmante no puede negar el hecho de que él firmo el documento.

A continuación se detallan las diferencias entre un esquema de firma hológrafo y uno digital:

Concepto	Firma Hológrafa	Firma Digital
Verificación de autenticidad de la firma	Un perito calígrafo puede determinar si la firma es fidedigna o ha sido imitada.	Mediante mecanismos criptográficos se puede verificar rápidamente si la firma es auténtica.
Verificación en la integridad del contenido	Muchas veces esta tarea puede ser ardua y llevar muchísimo tiempo de análisis. Otras veces, es imposible constatar si el documento firmado ha sufrido una leve modificación posterior a la firma.	La verificación de la integridad del documento es rápida y sencilla
Ubicación de la firma	La firma está asentada en el propio documento.	La firma digital no está embebida dentro del documento, sino por el contrario es una información periférica al documento. Es decir, el documento firmado no contiene la firma digital. Si bien son datos lógicamente separados, están fuertemente vinculados.

Concepto	Firma Hológrafa	Firma Digital
Terceros a confiar	Se confía en la pericia de los peritos calígrafos para resolver cualquier litigio	Dependiendo del esquema utilizado para las firmas digitales, existen terceros a los que se debe confiar como imparciales y ellos certifican la firma digital.
Velocidad de verificación	Baja. Requiere una verificación manual de los peritos	Instantánea

Tabla 10 – Comparación entre firma hológrafa y digital

En el esquema de firmas digitales con certificados X509 [9], una autoridad certificante es la que se encarga de constatar la identidad del firmante. La autoridad certificante es un tercero imparcial, la cual debe ser reconocida por ambas partes. La autoridad certificante emite certificados que contienen los datos básicos para identificar al firmante, función de *hashing* a utilizar y el algoritmo de cifrado, entre varios atributos más. Una descripción detallada del esquema de certificados X509 [9] escapa al alcance de este trabajo.

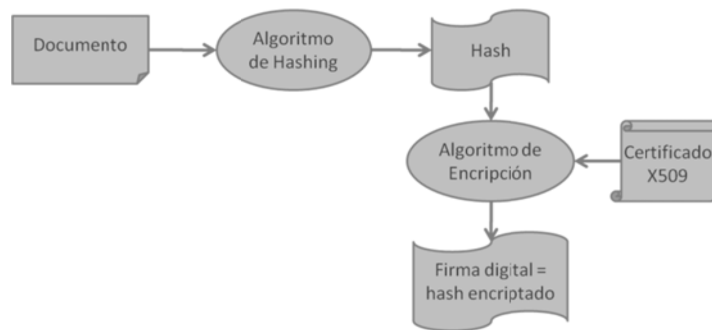


Ilustración 11 – Firma digital de documento usando un certificado X509

1.7. Colisión de *hash* con documento arbitrario

Debido a la existencia de colisiones en las funciones de *hashing*, existe la posibilidad de que exista un documento con el mismo *hash*. Si un atacante reemplaza el documento original con el nuevo documento, la adulteración de la información no sería detectada ya que los controles de integridad de la función de *hashing* fueron sobrepasados. Si el documento hubiese sido firmado, el documento duplicado por consiguiente también estaría firmado, aún sin el consentimiento del firmante. Indefectiblemente, este escenario representa un serio problema de seguridad.

En caso de encontrar una colisión cualquiera (sin una estructura lógica), el ataque de suplantación sería fácilmente detectable pues el

contenido del documento duplicado muy probablemente no sea legible ni respete ninguna semántica o formato. Si bien la colisión permitiría adulterar la información sin ser detectado, no tendría ningún otro provecho para el atacante, salvo el reemplazo y posible destrucción del documento original.

Ilustración 12 – Colisión con documento ilegible

Si un atacante quiere realizar un ataque más inteligente, podría poner como restricción un prefijo del documento con un contenido elegido que refleje lo que el atacante desea transmitir. El sufijo del documento (o también llamado *padding*) será manipulado para generar la colisión. Dicho *padding* casi con certeza tendrá un contenido ilegible puesto que estará conformado por bytes o caracteres sin un orden semántico.

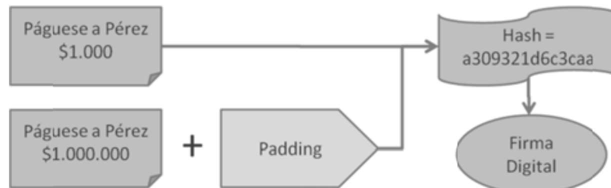


Ilustración 13 – Colisión con prefijo fijo

Para lograr un ataque perfecto de suplantación, el documento duplicado debería no sólo generar el *hash* deseado sino que su contenido debe ser elegido por el atacante y el *padding* debe respetar la semántica y formato del documento original. De esta forma, el documento duplicado tendrá el contenido que el atacante desea y al respetar la semántica del documento original la suplantación no levantará sospechas. La semántica de un documento dependerá del uso con el cual fue diseñado, ya sea un archivo de texto, ejecutable, una imagen o código fuente a compilar.

La forma de hallar estas colisiones con contenido inteligente reside en utilizar un prefijo establecido y un sufijo variable. El prefijo está compuesto por el contenido que el atacante desea que se transmita. El sufijo debe ser construido de forma inteligente para respetar la semántica del documento. El

padding será en efecto una cadena de datos secuenciales que permitirán ir barriendo distintos documentos modificando un símbolo por vez (carácter o byte según el caso).

De todos modos, estos tipos de ataques son muy difíciles de efectuar dado la inviabilidad computacional al buscar colisiones de *hash*. El espacio de documentos que generen una colisión de *hash* ya es bastante diminuta debido al diseño de las funciones de *hashing*. Este espacio se ve severamente reducido al agregar dos restricciones. Primero, el contenido deseado por el atacante es fijo y formará el prefijo del documento. Segundo, el *padding* generado deberá respetar una semántica. A medida que se agregan restricciones, el espacio de documentos se restringe, aumentando tanto la complejidad como el tiempo necesario para hallar la colisión.

A continuación se muestra un ejemplo en la confección de distintos documentos que colisionan, cada uno agregando ciertas restricciones. El primer tipo de colisión es un documento que, si bien colisiona, no tiene un contenido con un significado lógico, ni respeta la semántica del documento (en este caso es XML [2]). El segundo tipo de colisión contiene un prefijo con un significado, pero su *padding* sigue siendo ilegible. El tercer tipo de colisión es un documento con el significado deseado y con un *padding* semánticamente correcto y sin alterar el significado global del documento.

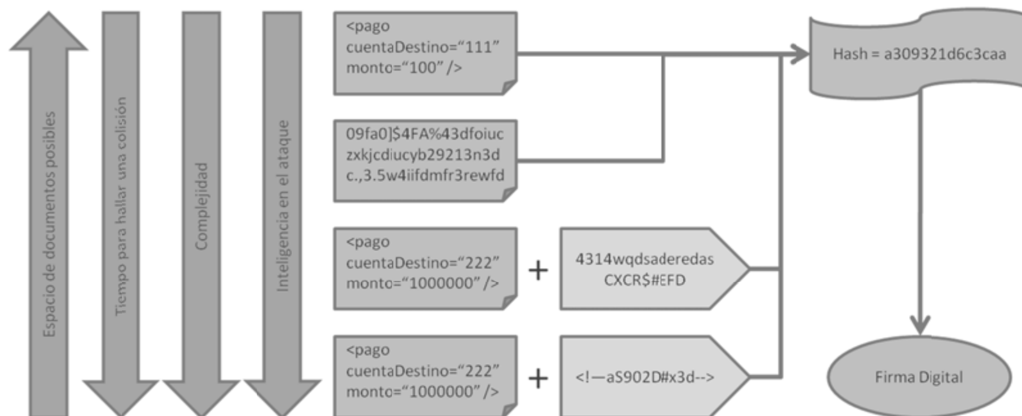


Ilustración 14 – Comparación de estrategias de colisón

1.8. Estado del arte

Previo al desarrollo de este trabajo, se ha investigado exhaustivamente y no se han encontrado herramientas similares. Si bien

existen proyectos o sistemas relacionados al área temática, ninguno se asemeja al enfoque que brinda HashCollider.

- Aplicaciones para quebrar un *hash*

En la actualidad, existen varios sistemas que almacenan las contraseñas según el *hash* respectivo, evitando así que se guarden en texto claro. Si un atacante toma conocimiento de un *hash*, en primera instancia no podrá acceder al sistema porque no conoce la contraseña que genera dicho *hash*. El atacante intentará realizar un *cracking* del *hash*, es decir, determinar qué contraseña generó dicho *hash*, para poder posteriormente ingresar al sistema.

Algunas herramientas que permiten efectuar el *cracking* de un *hash* son *Cain&Abel* [10], *John the Ripper* [11] y *l0phtCrack* [12]. Los ataques utilizan métodos como diccionarios con las palabras más utilizadas, permutaciones y expresiones regulares. Estos métodos permiten realizar ataques más eficientes que la fuerza bruta y en menor tiempo, pero sin la garantía de hallar un resultado.

- *Rainbow tables*

Existen varios proyectos que intentan generar una base de datos que permita mapear qué documentos generan cada *hash* para una función de *hashing* específica. El mecanismo es sencillo pero intensivo. Primero, se calculan *hashes* de una cantidad inmensa de documentos. Luego, se almacena el vínculo entre el *hash* y el documento que la origina en un índice gigante denominado *rainbow table* [13]. De esta forma, dado un *hash* conocido se podrá buscar en el *rainbow table* qué documentos lo originan. Debido a la cantidad de *hashes* posibles y los documentos asociados, las tablas son tan grandes que requieren de mucho espacio de almacenamiento.

El procesamiento de *hashes* es colaborativa, es decir, hay varios equipos que colaboran unos con otros para poder sumar un gran poder de cómputo. Este aspecto es similar al de HashCollider ya que también se requieren de varios equipos para la resolución de desafíos.

- Ataques avanzados a las funciones de *hashing*

Los ataques criptoanalíticos en las funciones *hash* son ataques que permiten reducir la complejidad o al menos predecir la salida. Para poder fabricar una colisión de *hash* de forma más eficiente, se deberá contar con una heurística o algoritmo para acotar el universo de posibles documentos a probar.

Existe un artículo muy interesante publicado por Marc Stevens, Arjen Lenstra y Benne de Weger en el cual exponen un vector de ataque para generar colisiones para un documento con prefijo fijo utilizando MD5 [14]. Los mismos autores han podido generar colisiones en archivos binarios [15] y también crearon certificados X509 [9] ilegítimos pero avalados por una autoridad certificante [16]. Stevens también ha publicado su tesis [28] incursionando en las colisiones de prefijo fijo y ha podido efectuar mejoras sustanciales a los ataques realizados por Wang.

- Ataque a MD5 con Flame

En 2012 se descubrió una serie de ataques dirigidos a ciertos países de África y el Oriente. Se utilizó un *software* llamado Flame [17] el cual utiliza una vulnerabilidad en el MD5 [4] que permite ataques de prefijo fijo. El ataque consistía en quebrar un certificado X509 [9] utilizado por Microsoft para firmar las actualizaciones de software y de esta forma diseminar código malicioso. Según se comenta, estarían utilizando la misma línea de investigación que publicaron Lenstra *et al* [14] mencionados anteriormente.

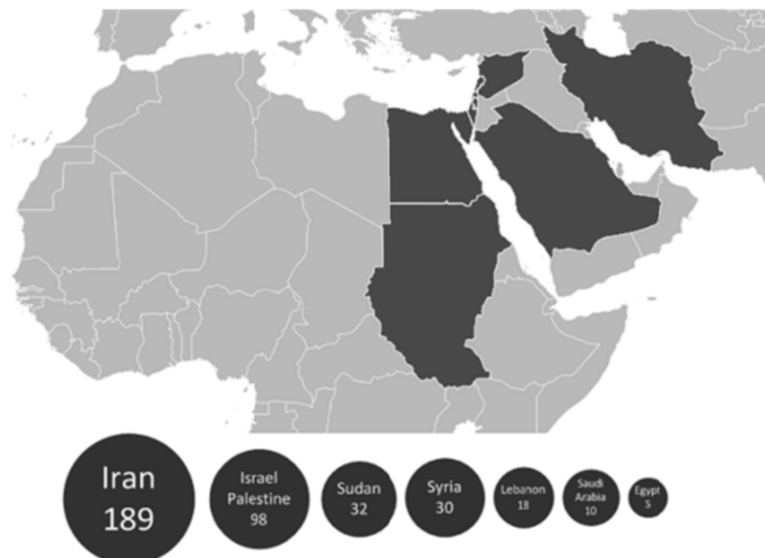


Ilustración 15 – Países afectados por ataque Flame a MD5 (Fuente: Kaspersky Labs)

2. Generación de colisiones con HashCollider

HashCollider es un sistema cuyo objetivo fundamental es una prueba de concepto para la resolución distribuida de colisiones de *hash* con documentos con un contenido escogido e inteligentemente creado. Los documentos fabricados respetarán los lineamientos semánticos según sea el tipo de documento que se desea colisionar. En esta sección se abarcarán todos los detalles sobre el diseño, construcción y utilización de la herramienta.

Para ver más detalles sobre los requerimientos funcionales del sistema, leer el Anexo B. En caso de desear profundizar sobre el diseño del software, por favor dirigirse al Anexo C. Por último, en el Anexo D se explica cómo un desarrollador podrá extender la funcionalidad del sistema.

2.1. Desafíos

Un desafío representa al conjunto de información necesaria para generar una colisión de *hash*. Se utiliza la palabra desafío para enmarcar este concepto ya que no se sabe cuándo se detectará una colisión y es una tarea a la cual habrá que procesar muchos documentos con la esperanza de hallar una colisión.

Un desafío incluye:

- El documento original al cual se desea duplicar
- El contenido que se desea que contenga la colisión
- La función de *hashing* a utilizar
- El tipo de desafío según las reglas semánticas que se deseen implementar.

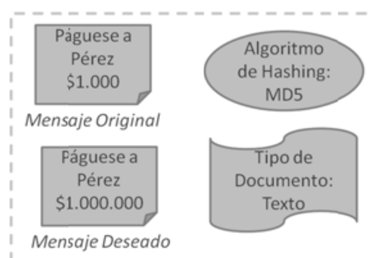


Ilustración 16 – Contenido de un desafío

2.2. Tipos de desafíos

Existen tres familias distintas de documentos en esta herramienta. Cada familia marca una semántica similar:

Familia	Descripción
Binario	Contenido binario, por ejemplo, un programa o una imagen. No es legible al ojo humano.
Texto	Texto libre sin formato. Sólo utiliza caracteres imprimibles. Es legible al ojo humano.
Código fuente	Es un caso particular de la familia de texto. Si bien es legible, debe cumplir con una sintaxis definida. Generalmente tienen una sintaxis reservada para los comentarios puestos por el programador para que se comprenda mejor el código. Los comentarios no afectan la funcionalidad del código fuente ya que al momento de precompilar (o ejecutar en caso de *scripts*) son sencillamente ignorados.

Tabla 17 – Familias de tipos de documento

HashCollider permite resolver desafíos de una amplia gama de tipos de documento. Los tipos de desafíos soportados son los siguientes:

Tipo de desafío	Familia
Bash	Código fuente
Binario	Binario
C	Código fuente
Haskell	Código fuente
HTML	Código fuente
Java	Código fuente
Perl	Código fuente
PHP	Código fuente
Python	Código fuente
Ruby	Código fuente
Texto	Texto
Visual Basic	Código fuente
XML	Código fuente

Tabla 18 – Tipos de Desafíos

2.3. Fragmentos

Para poder efectuar una resolución de forma distribuida, cada desafío será particionado en pequeños fragmentos a ser procesados de forma paralela. Un fragmento está compuesto por un listado de documentos candidatos a la colisión. Es imprescindible que los fragmentos sean disjuntos para evitar que haya documentos repetidos. La idea detrás de la fragmentación es poder delegar de forma coordinada dichos fragmentos a una gran cantidad de equipos que trabajen de forma paralela y distribuida.

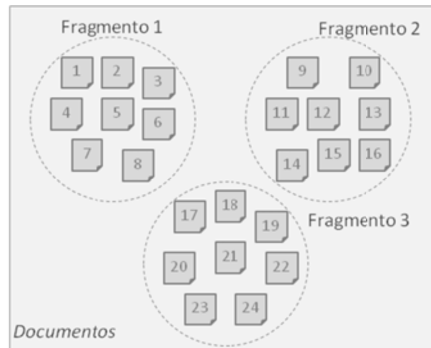


Ilustración 19 – Segmentación de desafío en fragmentos

La determinación de los documentos incluidos en un fragmento se efectúa al concatenar el contenido elegido por el atacante y el *padding* creado para cumplir con la semántica del documento. El *padding* está conformado por dos componentes:

- Un sufijo calculado por el motor de fragmentación. Cada tipo de desafío tiene asignada por diseño una cantidad de símbolos a procesar. A medida que se procesen documentos, el sufijo irá creciendo de tamaño y ocupando más símbolos.
- Un símbolo variable, análogo al byte menos significativo del documento candidato. El símbolo tomará todos los valores posibles según lo restrinja la semántica del documento a colisionar.

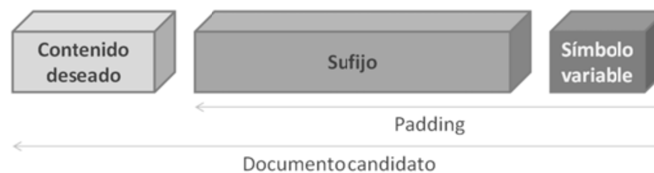


Ilustración 20 – Composición y armado de documento candidato

Los fragmentos son compactos y contienen toda la información necesaria para buscar una colisión entre los documentos asignados al fragmento. Al tener una estructura compacta, la transmisión de los fragmentos a los clientes será más rápida y aumentará la eficiencia en la resolución del desafío. Mucha de la información en un fragmento depende obviamente del desafío que se está resolviendo por lo que tiene muchos atributos en común.

Los componentes de un fragmento son:

- Identificador del desafío al que pertenece el fragmento. Este identificador será necesario en que se deba notificar el hallazgo de una colisión.
- Tipo de desafío para respetar las reglas semánticas
- Algoritmo de *hashing* para poder calcular los *hashes*
- El *hash* del documento original. Para efectuar la colisión no es necesario conocer el documento original, simplemente su *hash* asociado.
- El contenido deseado del documento suplantador.
- Sufijo secuencial para generar el *padding* al documento suplantador.

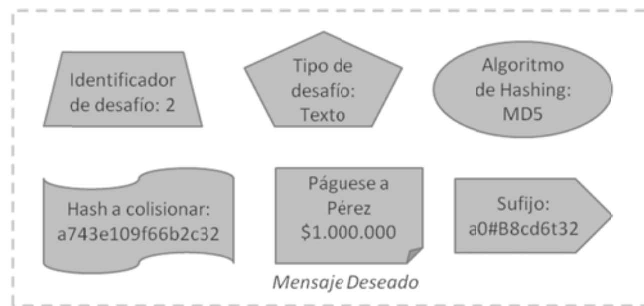


Ilustración 21 – Contenido de un fragmento

Cada fragmento en rigor conforma una partición del dominio de documentos. Esta propiedad es la que permite determinar que los fragmentos son independientes y disjuntos, permitiendo la resolución distribuida. Para que un fragmento sea considerado una partición, deberá cumplir las siguientes condiciones matemáticas. Sea A el conjunto de documentos posibles y A_i un fragmento:

- $\bigcup_{i \in I} A_i = A$: La unión de todos los fragmentos debe ser igual al espacio de documentos posibles
- $A_i \cap A_j \neq \emptyset \Rightarrow A_i = A_j$: La intersección entre cualquier fragmento es el conjunto vacío. Al utilizar un sufijo incremental, no existe la posibilidad de repetir documentos.
- $A_i \neq \emptyset$ para todo $i \in I$: Ningún fragmento debe estar vacío lo cual es bastante obvio ya que todo fragmento tendrá al menos un documento.

2.4. Diseño del sistema

Un nodo centralizado se encargará de fragmentar los desafíos y coordinar la resolución distribuida. Los clientes del HashCollider realizarán la ardua labor de resolver los fragmentos asignados en busca de una colisión.

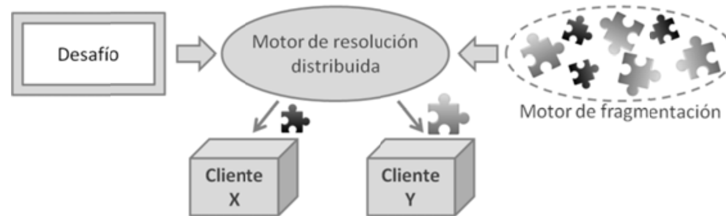


Ilustración 22 – Motor de fragmentación

La arquitectura está definida por las interacciones entre distintos componentes, cada uno cumpliendo una función específica.

Ilustración 23 – Arquitectura de HashCollider

A continuación se detallan los diferentes componentes:

- Aplicación *web*: Este sistema permite que cualquier persona pueda ver el estado y el avance de los desafíos publicados vía Internet. Además, un administrador podrá agregar nuevos desafíos.
- Servidor de coordinación: Es un *web service* que permite coordinar de forma centralizada la resolución distribuida de los desafíos. Los *web services* no requieren que sus clientes sean de una u otra plataforma, lo cual deja abierta la opción de poder construir otros clientes que no sean de Windows. El motor de fragmentación está incluido dentro de este componente. Para ver más detalles sobre los *web services*, se recomienda leer primero el Anexo A.

- Cliente para resolver colisiones: Es un servicio Windows que se ejecuta periódicamente, conectándose al *web service* y resolviendo los fragmentos delegados. En caso de hallar una colisión, se lo informará al *web service*.
- Configuración del cliente: Es otra aplicación del lado del cliente que le permitirá configurar cada cuánto resolverá un fragmento. Dado que la resolución de fragmentos consume recursos, la idea principal es que pueda resolver de forma periódica un fragmento, sin afectar el desempeño del equipo del cliente.

2.5. Plataforma

HashCollider fue desarrollado en una plataforma Windows utilizando el lenguaje de programación C# .*Net* 4.0 [7]. El sistema se ha diseñado utilizando un *web service* para brindar una ventaja a la hora de la portabilidad del sistema. La ventaja intrínseca de un utilizar un *web service* es que existen varios clientes que pueden consumir los métodos publicados, permitiendo trabajar de forma distribuida. Además, los clientes de un *web service* sólo requieren de la manipulación de XML [2] para el intercambio de parámetros de entrada y salida a cada método que se ejecute. Este requerimiento es muy fácil de complacer con cualquier sistema moderno y no impone restricciones en el software ni en el hardware. Debido a estas prestaciones, los clientes de un *web service* pueden coexistir y trabajar cada uno en plataformas distintas, ya sean de Windows, Unix, Linux u otras. Es más, a los ojos del *web service*, ni siquiera puede reconocer la plataforma de su cliente, ni necesita de procedimientos especiales que complejice su tarea.

La ventaja de trabajar a futuro con múltiples plataformas radica en que más clientes podrán ser desplegados a gran escala para trabajar de forma conjunta. Actualmente se desarrolló un cliente para Windows, pero en un futuro se podrá extender a otras plataformas.

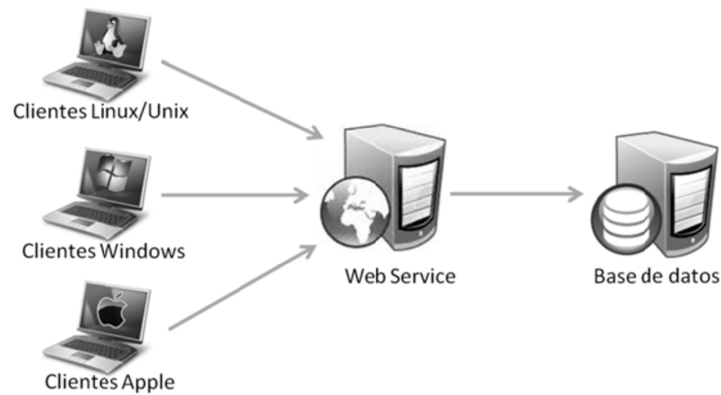


Ilustración 24 – Plataforma de clientes para el web service

2.6. Algoritmos de *hashing* utilizados

El sistema permite generar colisiones para un gran abanico de funciones de *hashing*. Las distintas familias de implementaciones de estos algoritmos se pueden observar fácilmente ya que muchas comparten un mismo prefijo.

- Haval128
- Haval160
- Haval192
- Haval224
- Haval256
- MD2
- MD4
- MD5
- RIPEMD128
- RIPEMD160
- SHA0
- SHA1
- SHA256
- SHA384
- SHA512
- Snefru
- Snefru256
- Tiger

Para poder integrar esta gran variedad de funciones de *hashing*, se utilizó una biblioteca de uso libre llamada ACryptoHashNet [18]. El mismo es un proyecto de código abierto cuyo objetivo es brindar las implementaciones de distintos funciones de *hashing* para ser utilizados en el ambiente de .*Net* [7].

2.7. Motor de fragmentación

El motor de fragmentación contenida dentro de HashCollider es el cerebro que orquesta la resolución distribuida de los desafíos. La misma se encarga de fraccionar el desafío en pedazos pequeños y asignarlos a cualquier cliente que así lo requiera.

Los objetivos principales del motor son el alto rendimiento, alta concurrencia y un tiempo de respuesta mínimo. Para poder lograr un alto volumen de fragmentos, el servidor que sincroniza la resolución fue construido para ser rápido y simple. La tarea de asignar un fragmento a un cliente es escalable ya que no se requiere una gran cantidad de soporte digital para almacenar el estado de cada desafío.

Los fragmentos asignados por HashCollider son contiguos, es decir, están estrechamente relacionadas mediante el sufijo variable que conformará el *padding*. Al ser contiguos, el cálculo de fragmentos es sencillo y rápido, pero más importante aún, son disjuntos por lo que no se repetirán documentos a probar.

El motor de sincronización se basa en la premisa de no guardar un histórico de los fragmentos asignados a cada cliente ya que esto consumiría muchos recursos. Una vez asignado el fragmento, HashCollider asume que ese fragmento no tiene una colisión y descarta el fragmento. En caso contrario, el cliente notificará la colisión. Esta característica permite que sólo se guarde el último fragmento lo cual reduce mucho el espacio de disco requerido para almacenar la sincronización de fragmentos y acelera la capacidad operativa. Al asignar un nuevo fragmento, el motor simplemente necesita actualizar el estado del próximo fragmento.

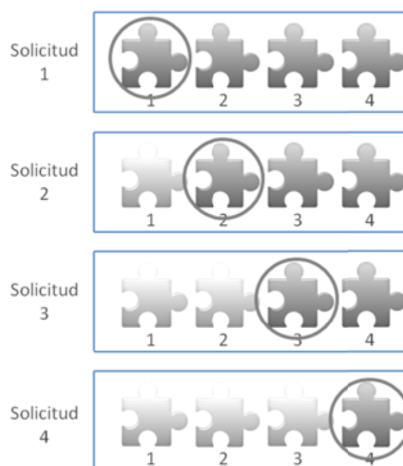


Ilustración 25 – Fragmentación correlativa

2.8. Resolución de fragmentos

Para poder resolver un fragmento, el cliente deberá recorrer los documentos asociados y calcularles el *hash* asociado. Cada tipo de desafío tiene configurado la cantidad de permutaciones a realizar en el sufijo para un fragmento específico. El *padding* crecerá indefinidamente siempre generando documentos distintos y únicos. Para cada fragmento, el cliente deberá recorrer todas las posiciones posibles del símbolo variable (el último símbolo del *padding*) e ir desplazando el sufijo variable en una posición.

En las siguientes líneas se analizará un caso práctico. Si se toma el conjunto de símbolos posibles para un *padding* como S = {a-z, A-Z, 0-9}, el cardinal de dicho conjunto es de 62 elementos. Para cada valor del sufijo variable, se estarán procesando 62 documentos distintos y únicos. El sufijo variable actúa como un contador. Cada 62 documentos procesados, el sufijo variable se actualiza con el siguiente símbolo. El sufijo variable por ejemplo empezará con el valor "a", luego con "b" y así siguiendo. Al acabarse los símbolos representados en el símbolo menos significativo, el sufijo variable aumenta en tamaño en un símbolo. Por ejemplo, al terminar de procesar el sufijo "9", el sufijo se desplaza a "aa".

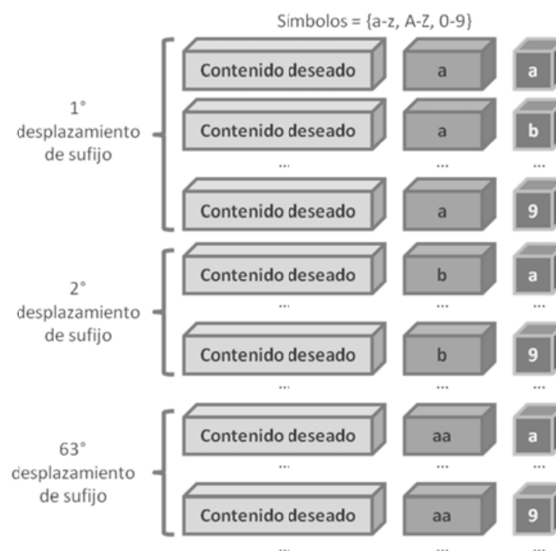


Ilustración 26 – Documentos procesados en un fragmento

Al asignar un nuevo fragmento, HashCollider automáticamente incrementará la cantidad de documentos procesados asociados a dicho fragmento. Esta actualización constante permitirá publicar el estado de la

resolución con datos recientes y para que pueda ser visualizado desde el sitio *web*. La fórmula para calcular la cantidad de documentos asociados a un fragmento es multiplicar la cantidad de permutaciones a efectuar en el sufijo por el número de símbolos posibles según la semántica del tipo de documento.

2.9. Estrategias en el armado de *padding*

Cada tipo de desafío tiene una estrategia de resolución particular. La semántica acorde al tipo de documento impondrá restricciones en el armado del *padding*. La estrategia está ligada a la familia de tipos de documento (binario, texto o código fuente) a la que pertenezca el desafío.

Familia	Estrategia	Tipo de Símbolos	Símbolos
Binario	Agregar bytes al final del documento	Byte	{0 – 255}
Texto	Agregar caracteres blancos al final del documento	Carácter	Ej.: {<SPACE>, <ENTER>, <TAB>}
Código fuente	Agregar un *padding* al final del documento encapsulado con la sintaxis de comentarios de programador	Carácter	Depende de los caracteres utilizados para denotar el comienzo y fin de comentarios en cada lenguaje de programación Ej.: {a-z, A-Z, 0-9,+,-,[,],(,)}

Tabla 27 – Estrategias de armado de documentos candidatos

Los documentos binarios son los menos restrictivos. Los archivos ejecutables tienen una particularidad por la cual se pueden agregar bytes al final del documento y el comportamiento original del programa no es afectado en lo absoluto. Al ser un archivo binario se podrá utilizar un sufijo sin restricciones en los símbolos a utilizar por lo que se podrán usar los 256 valores posibles de cada byte. Este es el caso más sencillo, ya que el *padding* en estos casos no debe respetar ninguna semántica dado que no altera el flujo de ejecución del programa.

Los documentos de texto son los más restrictivos. La restricción es implícita ya que el texto debe ser legible por un ser humano por lo que no se puede utilizar un *padding* que no forme palabras y no tenga un hilo conductor. Por consiguiente, los símbolos a utilizar en los documentos de tipo de texto son los denominados caracteres blancos, los cuales son imprimibles pero son invisibles, tal como el espacio o la tabulación. Para realizar una fragmentación sencilla y rápida, el *padding* se realizará al final

del documento ya que si se deseara agregar espacios en blanco en distintos lugares del documento éste se deformaría y requeriría un seguimiento y coordinación mucho más compleja y lenta.

Los documentos de código tienen un grado intermedio de restricción. La ventaja de este tipo de documentos es la utilización de los comentarios del programador que no afectan el código fuente. Dentro de los comentarios se podrán agregar una gran variedad de caracteres imprimibles siempre y cuando no se afecte la delimitación de los comentarios. La sintaxis para los comentarios de los distintos tipos de lenguajes se detalla a continuación:

Tipo de desafío	Comentarios
Bash	<#comentarios#>
C	/*comentarios*/
Haskell	{-comentarios-}
HTML	<!--Comentarios-->
Java	/*comentarios*/
Perl	#comentarios
PHP	/*comentarios*/
Python	<#comentarios#>
Ruby	#comentarios
Visual Basic	‘comentarios
XML	<!--comentarios-->

Ilustración 28 – Sintáxis de comentarios para distintos lenguajes de programación

2.10. *Buffer Caché* en el motor de fragmentación

El motor de fragmentación deberá responder ágilmente a muchas consultas concurrentes provenientes de todos los clientes que estén resolviendo el desafío. Para poder brindar un servicio de bajo tiempo de respuesta, se ha incorporado un módulo de acceso centralizado a los datos, denominado *buffer caché* [19].

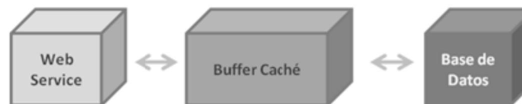


Ilustración 29 – Buffer Caché

El *buffer caché* es un gestor del acceso a los desafíos del sistema. Cualquier consulta o modificación de un desafío deberá pasar únicamente por el *buffer caché*. Este mecanismo permite administrar cuándo un desafío reside en memoria, logrando un acceso más rápido y evitando la penalización de una consulta a la base de datos. Conjuntamente, deberá administrar cuándo serán persistidos los cambios efectuados en un desafío

ya que la escritura es diferida, es decir, no es sincrónica. Se pueden hacer varias modificaciones al estado del desafío (por ejemplo, un desplazamiento al próximo fragmento) y todas se realizarían en memoria.

El objetivo principal de este módulo es disminuir las consultas a la base de datos, las cuales penalizan el desempeño de la herramienta. El *web service* efectúa constantemente consultas a la base de datos para poder calcular el siguiente fragmento del desafío y almacenar el estado actual de la resolución al efectuar el desplazamiento en el sufijo. Para cada uno de los fragmentos pedidos por los clientes, el *web service* deberá realizar una y otra vez las mismas consultas. El *buffer caché* permitirá acelerar la gran mayoría de las tareas repetitivas del motor de fragmentación, minimizando los recursos utilizados, disminuyendo el tiempo de respuesta y aumentando el caudal de fragmentos que se pueden asignar.

El *buffer caché* permitirá brindar un servicio más eficiente, tolerando mayores niveles de *stress* debido a una alta concurrencia de clientes. Por ejemplo, si el buffer caché está configurado para efectuar la escritura cada 100 consultas, al haber finalizada dicha cantidad de fragmentos se habrá hecho una sola consulta a la base de datos (y su respectiva carga de datos en memoria) y una sola escritura. En este caso, se efectuó una aceleración algorítmica [20] de 100 en estas dos tareas. Es decir, las dos tareas principales del *web service*, que a su vez son las más frecuentemente procesadas, se ha reducido su tiempo de ejecución al 1%.

En caso de hallar una colisión o que el administrador seleccione otro desafío a resolver, se le enviará un mensaje al *buffer caché* para que almacene su estado de forma permanente en la base de datos. De esta forma, estas modificaciones críticas se persistirán enseguida y el buffer caché quedará vacío y listo para cargar un nuevo desafío.

2.11. Desplazamiento de fragmentos en clientes

HashCollider tiene establecido un protocolo de comunicación. Para ver más detalles sobre cómo se implementa esta mejora o para ahondar en cuáles son los métodos del *web service* o cómo es el protocolo de invocaciones, por favor dirigirse al Anexo E.

Cada vez que un cliente de HashCollider pide un fragmento a resolver, el motor de fragmentación le deberá transmitir toda la información del desafío necesaria para hallar una colisión. Si bien este enfoque es efectivo, la eficiencia es bastante baja ya que el motor de fragmentación deberá enviar repetidas veces información muy similar. El desempeño en la resolución distribuida tendrá un serio detrimento cuando el documento a resolver tenga un gran tamaño.

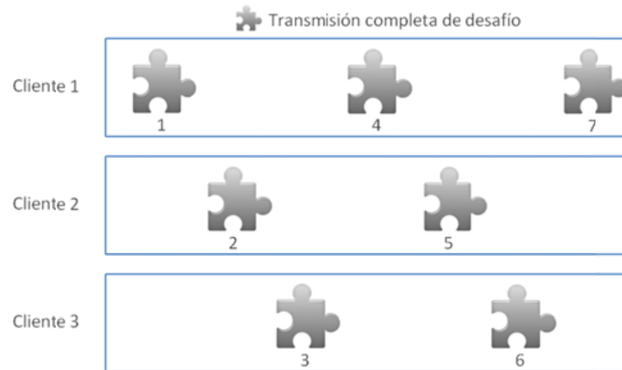


Ilustración 30 – Transmisión completa de fragmentos

Cuando un cliente está resolviendo un desafío, los datos que varían entre un fragmento y otro son muy escasos. En particular, el único dato que varía es el sufijo secuencial. Sin embargo, según el enfoque anterior que involucraba la transmisión completa del fragmento, el cliente deberá cargar en memoria todos los datos necesarios para cada fragmento que descargue. HashCollider provee un segundo enfoque optimizado para reducir estas transferencias innecesarias y aumentar el desempeño de la herramienta. Los clientes podrán efectuar un desplazamiento lógico en memoria. La descarga completa de datos del fragmento se realizaría una única vez.

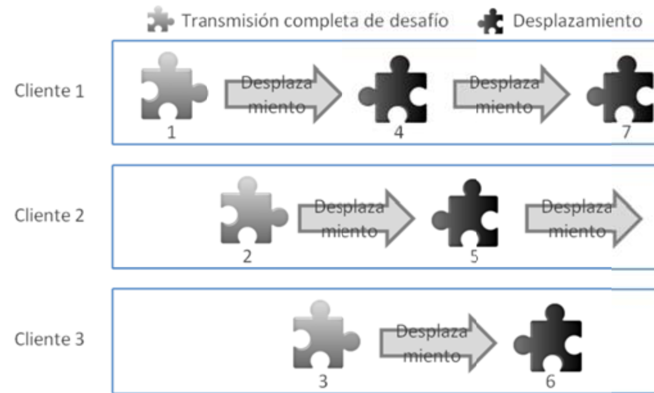


Ilustración 31 – Transmisión y desplazamiento automático de fragmentos

2.12. Pruebas integrales

¿Cómo se puede determinar si HashCollider efectivamente funciona? Se realizó una serie de pruebas integrales que puedan asegurar la efectividad en la resolución de colisiones. Estas pruebas se ingeniaron para que la colisión se obtenga con poco procesamiento.

El secreto de crear unos desafíos de prueba es utilizar un documento original idéntico al documento deseado, pero además al documento original se le agregó un *padding* especial según la semántica del tipo de desafío. Cuando HashCollider procese distintos documentos y justo pase por este *padding* escogido deliberadamente, el *hash* será el mismo al documento original y la colisión es inmediata. En resumen, el documento original debe ser uno de los documentos que HashCollider procesará.

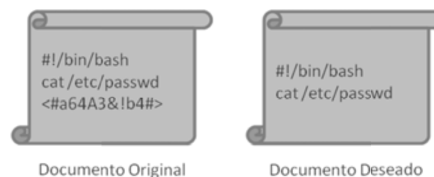


Ilustración 32 – Diseño de archivos de prueba

Se crearon desafíos con estas pruebas para generar colisiones rápidamente. En la primera fase de pruebas se evaluó si el sistema hallaba una colisión en todos los tipos de documentos disponibles.

La segunda fase de pruebas fue más rigurosa teniendo que procesar millones de documentos para hallar la colisión.

La tercera y última fase de pruebas tuvo como objetivo evaluar la resolución distribuida. Las primeras dos fases se ejecutaron en un solo equipo. Para poder realizar esta última fase de pruebas, se instalaron varios clientes HashCollider en distintos equipos conectados en una misma red. El sistema pudo resolver efectivamente los desafíos de forma colaborativa.

En resumen, las tres fases de pruebas se detallan a continuación:

Fase	Prueba	Documentos procesados	Tipos de desafío probado	¿Distribuido?	Objetivo
1	Funcional	Cientos	Todos	No	Detectar si HashCollider resuelve todos los tipos de desafíos disponibles.
2	Volumen	Decenas de millones	Muestreo	No	Detectar si HashCollider efectúa una correcta fragmentación y resolución de fragmentos de forma extensiva.
3	Coordinación	Decenas de millones	Muestreo	Si	Detectar si HashCollider resuelve las colisiones de forma distribuida.

Ilustración 33 – Pruebas realizadas

Dentro del paquete de aplicaciones desarrolladas, se incluyó una herramienta diseñada para la depuración de errores. La misma es una aplicación de escritorio que permite descargar de forma manual un fragmento y resolverlo. Una vez descargado, la aplicación imprime en una bitácora en pantalla todos los datos asociados al fragmento incluyendo el nombre del método invocado del *web service*. Esta herramienta resultó de gran valor durante la etapa de desarrollo de HashCollider y facilitó la identificación de fallas en el *software* y su posterior corrección.

Por supuesto, se han efectuado pruebas de resolución de desafíos reales (sin aplicar a la técnica de aplicar el mismo prefijo en el documento original y en el deseado). No se han tenido hasta la fecha pruebas fehacientes ni un caso de éxito de colisión. Será cuestión de tiempo y de conseguir suficiente poder de cómputo distribuido para llegar a un caso de éxito.

2.13. Publicación web del avance

El sistema cuenta con un sistema web que permite a cualquier persona en el mundo ver el estado actual de la resolución de los desafíos. Para más detalles, leer el Anexo F.

Métricas de desempeño

En esta sección se analizó el funcionamiento de la herramienta para determinar ciertas características y patrones sobre su utilización. Todas las pruebas se realizaron utilizando el mismo equipo.

Entorno de ejecución

Todas las métricas fueron medidas en un sólo equipo con Windows 7 Home Premium [21] de 64 *bits* con un microprocesador Intel Core i5 [22] de 2,5GHz y 4GB de memoria dinámica. Se esperaría un desempeño mayor si se utilizaran más equipos.

El equipo procesaba el servidor *web*, el *web service*, la base de datos y el cliente. Es decir, los recursos del equipo fueron compartidos entre todos los procesos. Obviamente, si se utilizaran recursos dedicados en distintos equipos el desempeño debería ser más alto.

Además, todos los sistemas se ejecutaron en modo de depuración utilizando la interfaz de desarrollo del Visual Studio 2010 [23]. Si se ejecutaran los distintos componentes en modo optimizado el rendimiento sería aún mayor.

Caudal de documentos procesados

Se creó un desafío complejo a ser resuelto luego de un par de millones de documentos procesados. Luego se promedió la duración total de la resolución para luego calcular la cantidad de documentos procesados por segundo. En este caso, se utilizó un desafío que colisionaría al procesar 2.250.240 documentos y la duración en promedio fue de unos 31 segundos. Según estos datos, el caudal de documentos procesados asciende a unos 72.588 documentos por segundo utilizando un único cliente (también llamado nodo).

Escalabilidad

Dadas las métricas computadas para este sistema, es posible estimar la escalabilidad de esta herramienta para un ataque real a gran escala. En

tal caso, se deberá evaluar la eficiencia del sistema según la cantidad de clientes (también llamados nodos) que estén procesando de forma paralela.

Para poder efectuar estas estimaciones es necesario definir el escenario a estudiar. Se tomará como aproximación pesimista del caudal del sistema a los 72.000 documentos por segundo por nodo calculados previamente. Además, se utilizará a la función de *hashing* MD5 [4] que genera *hashes* de 128 *bits*. Un ataque de cumpleaños [1] para hallar una colisión reducirá el espacio de documentos a $2^{m/2}$, siendo m el tamaño del *hash*. En el caso de MD5 [4], se hallará una colisión al procesar 2^{64} documentos.

Según una cantidad conocida de nodos, se podrá estimar la duración del ataque al dividir la cantidad total de documentos a procesar por la multiplicación del caudal y la cantidad de nodos.

Nodos	Duración de ataque (años)
1	8.124.200
2	4.062.100
4	2.031.050
8	1.015.525
16	507.763
32	253.881
64	126.941
128	63.470
256	31.735
512	15.868
1.024	7.934
2.048	3.967
4.096	1.983
8.192	992
16.384	496
32.768	248
65.536	124
131.072	62,0
262.144	31,0
524.288	15,5
1.048.576	7,75
2.097.152	3,87
4.194.304	1,94
8.388.608	0,97
16.777.216	0,48

Tabla 34 – Duración de un ataque a MD5 según cantidad de nodos

Tal como se puede observar, con estas aproximaciones de alto nivel se requerirá un total de alrededor de 17 millones de clientes para hallar una colisión en aproximadamente 6 meses. Recordar que la eficiencia podría ser considerablemente mayor si no se tuviesen las restricciones identificadas en el ambiente de pruebas.

Si bien la escalabilidad actual de la herramienta es bastante baja, se podría efectuar una estimación de cuál sería su escalabilidad en el caso de que se pueda implementar ataques criptoanalíticos basados en prefijos fijos. En el caso del trabajo realizado por Stevens *et al* [28] se puede apreciar que sus ataques a MD5 logran mejorar el ataque de cumpleaños [1] del orden de 2^{64} documentos a un increíble 2^{50}. Si se pudiera adaptar las técnicas criptoanalíticas descritas por Stevens y embeberlas dentro del sistema HashCollider la escalabilidad se reduciría drásticamente. Basta con realizar algunos cálculos tomando el mismo caudal de documentos que procesa el sistema (72000 documentos por segundo).

Nodos	Duración de ataque (años)
1	495,86
2	247,93
4	123,97
8	61,98
16	30,99
32	15,50
64	7,75
128	3,87
256	1,94
512	0,97
1.024	0,48
2.048	0,24
4.096	0,12

Tabla 35 – Duración de un ataque a MD5 utilizando el enfoque de Stevens

Según los resultados de la escalabilidad utilizando el enfoque propuesto por Stevens *et al*, el sistema sería capaz de quebrar un MD5 con un documento semánticamente correcto y con el prefijo deseado en tan solo 44 días utilizando 4096 nodos. Esta nueva eficiencia demostraría ser un salto gigante en crear este tipo de colisiones.

Conclusiones

La construcción de HashCollider brinda un enfoque nuevo y práctico en un área poco explorado en la seguridad informática. La posibilidad de crear colisiones de *hash* con documentos que literalmente contengan el contenido deseado tiene una gran implicancia en el resguardo de documentos. Un ataque de este estilo vulnera los esquemas de control de integridad basados en *hashes* y la implementación de firmas digitales.

Esta herramienta ofrece un aporte innovador para estos tipos de ataques. Si bien hallar un documento que colisione con un *hash* y además respete ciertas reglas semánticas es una tarea ardua, HashCollider permite resolver estos desafíos de forma distribuida. La herramienta utiliza como plataforma de lanzamiento a la poderosa expansión de Internet y la conectividad lograda a nivel mundial. Millares de clientes distribuidos en todo el mundo se podrán agrupar y unir fuerzas para conformar un alto poder de cómputo distribuido y coordinado con un objetivo en particular.

Es importante remarcar de forma terminante que HashCollider es simplemente una prueba de concepto. HashCollider simplemente explicita un enfoque de cómo encarar esta problemática, la cual expresa el aporte del autor. Existen muchas mejoras que se podrían realizar a la herramienta para aumentar su efectividad.

Pruebas de la hipótesis

Tal como se la presentó en el capítulo introductorio, la hipótesis de este trabajo es la siguiente:

HashCollider es una herramienta que facilitará la generación de colisiones de *hash* en base a un documento de contenido arbitrario y escogido, respetando las reglas semánticas según el tipo de documento.

Para validar dicha hipótesis se procederá a desmenuzar las frases que componen dicha afirmación.

- "HashCollider es una herramienta…": HashCollider fue concebido y diseñado como un software cuyos resultados ofrecen un enfoque innovador. Como toda herramienta, se la podrá mejorar y adaptar en el futuro para hacerla más eficiente. El *software* podrá ser descargado

libremente por cualquier persona en el mundo. El mero alcance de este sistema es ofrecer una prueba de concepto que quizás actúe de disparador para nuevas ideas en la comunidad criptográfica.

- "… que facilitará la generación de colisiones de *hash*…": HashCollider es un sistema desarrollado para la resolución distribuida de colisiones de *hashes*. La idea principal de este sistema es aportar una solución novedosa e innovadora para poder efectuar el procesamiento de forma armoniosa y eficiente. Al contar con la capacidad de coordinar la resolución de forma distribuida, se podrán utilizar varios equipos que colaboren resolviendo pequeños fragmentos de forma paralela. Según las pruebas realizadas, el sistema funciona correctamente y se puede afirmar que el sistema es capaz de hallar una colisión aunque esto implique procesar millones de documentos.
- "… en base a un documento de contenido arbitrario y escogido…": HashCollider permite configurar cada desafío publicado, debiendo escoger el contenido que conformará el documento a colisionar. La colisión no será generada mediante un documento que no tenga ningún significado interpretable (ya sea por el ser humano o por máquinas), sino todo el contrario: la idea es forzar una colisión con un sentido y significado intencionalmente.
- "respetando las reglas semánticas…": Los documentos forjados en esta herramienta que generen las colisiones de *hash* no sólo tendrán un sentido intencional, sino que se respetará un formato y las reglas semánticas pertinentes en toda la extensión del documento.
- "… según el tipo de documento": HashCollider contempla una gran variedad de formatos de documentos. Cada formato tiene sus reglas semánticas asociadas. El proceso de fragmentación y manipulación del *padding* deberá ser acorde al tipo de desafío asignado.

La pregunta primordial es ¿esta herramienta efectivamente funciona tal como se espera que funcione? Para poder disipar toda duda, se diseñó y documentó una batería extensiva de pruebas para verificar el funcionamiento. Según los resultados de dichas pruebas, se puede afirmar que la herramienta está en condiciones de funcionar de forma concurrente en varios equipos interconectados y procesar de forma coordinada millones

de documentos hasta hallar la colisión. En pocas palabras, si existe una colisión la herramienta la hallará.

Mejoras futuras y posibles líneas de profundización

La utilización de *web services* en HashCollider permite trabajar en múltiples plataformas para poder abarcar una mayor cantidad de equipos a gran escala. Tal como se mencionó en la sección que detalla el diseño de la herramienta, los clientes instalados en diversas plataformas podrán coexistir. El alcance de este trabajo es una prueba de concepto y sólo se desarrolló un cliente para la plataforma Windows. Se podrían desarrollar clientes para otras plataformas, como Linux o Unix, y así facilitar la divulgación y colaboración para la resolución de los desafíos.

Otra línea de mejora que involucra un nivel de investigación mucho más profundo, es el de introducir mejoras en la utilización de vulnerabilidades criptoanalíticas en las funciones de *hashing*. Si bien existen algunas vulnerabilidades en MD5 [4] utilizando un prefijo conocido, el algoritmo para capitalizar dichas vulnerabilidades no es de público conocimiento. Estos ataques podrían apalancar la generación de heurísticas para aumentar significativamente la escalabilidad y eficiencia del sistema.

La idea de poder incorporar una heurística inteligente que permita reducir la cantidad de documentos a procesar es muy tentadora. Principalmente, se deberá obtener la documentación necesaria para reproducir estos ataques. Segundo, se deberá investigar y comprender dichos ataques y analizar la forma de fragmentar los documentos candidatos. No se debe perder de vista el concepto de respetar las reglas semánticas. Tercero, se debe programar esta nueva funcionalidad e integrarla a la herramienta y capitalizar su potencial. Si alguno de estos ataques fuese posible de incorporar, la eficiencia de la herramienta se elevaría considerablemente.

Otra línea de mejora o modificación sería crear tipos de desafíos más complejos y utilizados en la actualidad. Se podrían tomar las estructuras para transferencias bancarias, pagos con tarjeta de crédito, certificados X509 o cualquier otra transacción utilizada en los negocios.

Escalabilidad y efectividad

El sistema es simplemente una prueba de concepto y aún no resolvería un desafío en tiempos y recursos aceptables. Se ha estimado que para quebrar un hash MD5 harían falta 6 meses y 17 millones de clientes procesando. Esta estimación es pesimista pero se podría mejorar utilizando recursos dedicados para este proyecto. Vale la pena recordar que el procesamiento de documentos es en realidad a fuerza bruta.

Si se pudiese incorporar las heurísticas propias de los ataques de prefijo fijo de Stevens *et al* [28], la escalabilidad sería muchísimo más alta. Se estima que se podría quebrar un *hash* MD5 en 44 días utilizando sólo 4000 equipos.

Anexo A: *Web Services*

En este anexo se explicará a grandes rasgos cómo funciona un *web service* [24].

Definición

Un *web service* es un *software* diseñado para la interacción entre distintos equipos conectados mediante una red. El *web service* publica a su vez una interfaz para el acceso a ciertos métodos, cada uno con una serie de parámetros de entrada y salida correspondientes. Dicha interfaz está especificada utilizando un formato conocido como WSDL (*Web Services Description Language*).

La interacción con el web service utiliza como fundamento otras tecnologías ampliamente utilizadas como XML [2], *Hypertext Transfer Protocol* (HTTP) [25] y *Simple Object Access Protocol* (SOAP) [26].

Propósito

Un *web service* es una tecnología simple y fácil de usar que permite a un *software* invocar métodos que se ejecutan de forma remota. Los métodos remotos se publican mediante un *web service*. Como toda función, existirán parámetros de entrada y salida correspondientes. Dichos métodos pueden ser utilizados en una gran variedad de plataformas y sistemas operativos lo cual garantiza interoperabilidad. El objetivo principal de los *web services* es ofrecer una interfaz de acceso a un sistema y permitir la integración con otros sistemas periféricos.

Anexo B: Requerimientos del Sistema

Los requerimientos funcionales de alto nivel efectuados en HashCollider se detallan a continuación:

- Motor de sincronización:

1. Implementado en un *web service* (ver Anexo A) para hacer posible la resolución remota y portable a distintas plataformas.

2. El *web service* deberá implementar los métodos definidos en el protocolo de comunicación definido en el Anexo E.

3. Deberá contemplar una variedad de tipos de desafíos, como archivos binarios, de texto y de código fuente.

4. Deberá manejar una variedad de funciones de hashing, incluidas MD5 y SHA1.

5. Deberá poder fragmentar de forma disjunta pequeños bloques del desafío para permitir la resolución distribuida.

6. La fragmentación debe ser rápida y sólo se almacenará el estado del fragmento actual en ejecución. No se almacenará un histórico de fragmentos.

7. Al hallar una colisión, se deberá almacenar el documento que generó dicha colisión, el día y la hora del descubrimiento y la cantidad de documentos procesados al momento.

8. Al hallar una colisión se enviará un email a la lista de distribución.

- Página *web*:

9. Se podrá visualizar el estado de resolución de los desafíos.

10. Se podrá descargar el software para instalar el cliente de HashCollider.

11. Cualquier persona se podrá suscribir a la lista de distribución de correos electrónicos para recibir novedades.

12. Cualquier persona suscrita a la lista de distribución podrá solicitar ser dado de baja.

13. El administrador deberá ingresar mediante una autenticación con usuario y contraseña.

14. El administrador podrá agregar, modificar o eliminar desafíos.
15. El administrador podrá cambiar la prioridad de los desafíos.

- Cliente:

16. Desarrollar un sistema para automatizar la solicitud y resolución de fragmentos. En el caso de Windows será un servicio.
17. Deberá implementar el protocolo de comunicación con el *web service* del motor de fragmentación especificado en el Anexo E.
18. Se deberá generar un instalador para el despliegue del producto en varios clientes.
19. El usuario podrá configurar con qué frecuencia el cliente resolverá fragmentos para restringir la cantidad de recursos que serán consumidos.

Los requerimientos no funcionales se detallan a continuación:

- Base de datos SQL
- Microsoft .Net Framework 4.0
- Visual Studio 2010
- Equipo con más de 3GB de memoria RAM y con un procesador rápido (al menos dos núcleos)

Anexo C: Diseño del Sistema

En esta sección se detallarán los aspectos del diseño del sistema utilizando las herramientas que ofrece UML (*Unified Modeling Language*) [27].

Diagrama de Despliegue

Este tipo de diagramas permite identificar los nodos (aplicaciones) que interactuarán al momento de desplegar la herramienta y colocarla en un ambiente productivo.

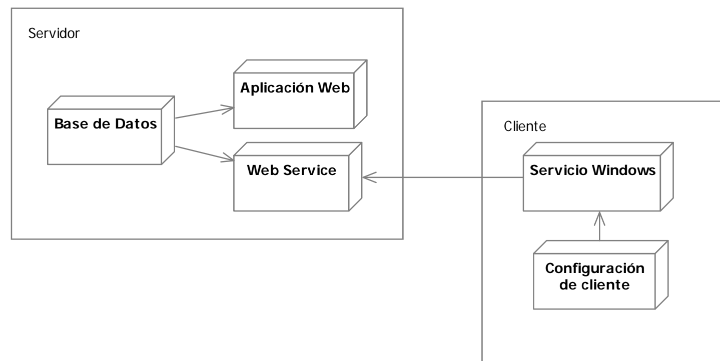


Ilustración 36 – Diagrama UML de despliegue

Se pueden distinguir fácilmente dos grupos de nodos: el servidor y el cliente. El servidor cuenta con el *web service* para coordinar la resolución distribuida, la aplicación *web* para la consulta en línea de las estadísticas y la base de datos para almacenar las transacciones. El cliente cuenta con un servicio Windows que permite ejecutar de forma automatizada la tarea repetitiva de resolver fragmentos y una aplicación que permitirá configurar con qué frecuencia se resolverán los fragmentos.

Diagrama de Procesos

El diagrama de procesos permite identificar cómo interactúan los distintos procesos que se ejecutarán y cuál es el intercambio de datos. En el caso de la solicitud de un fragmento por un cliente, el proceso del servicio de Windows invocará un método del *web service* para obtener los datos necesarios para resolver el siguiente fragmento. El *web service* a su vez

buscará el estado del desafío utilizando el motor de la base de datos, calculará el próximo fragmento, almacena el nuevo estado y devuelve al servicio Windows los parámetros para el resolver el nuevo fragmento.

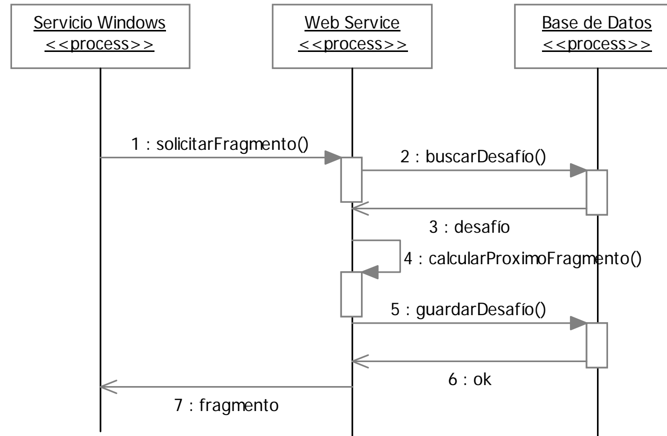


Ilustración 37 – Diagrama UML de procesos

Diagrama de Componentes

Un componente es el resultado obtenido al compilar código fuente agrupado en un una aplicación o en una biblioteca según sea el caso. Los distintos componentes tienen dependencias entre sí al utilizar cierta funcionalidad.

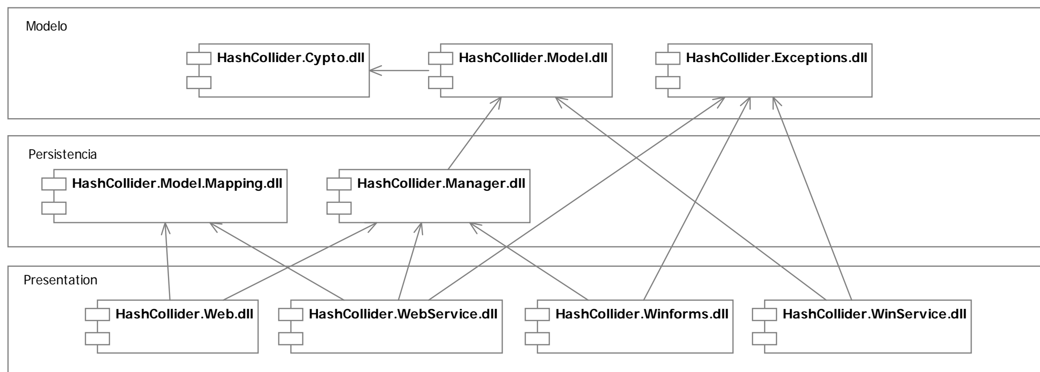


Ilustración 38 – Diagrama UML de componentes

Los componentes dentro del sistema se detallan a continuación:

- HashCollider.Crypto.dll: Encapsula toda la funcionalidad para utilizar distintas funciones de *hashing*.

- HashCollider.Model.dll: Contiene todas las entidades de datos modelados y sus respectivas relaciones. En especial, contiene toda la lógica para resolver fragmentos para cada tipo de desafío.
- HashCollider.Exceptions.dll: Contiene todas las excepciones involucradas en los métodos del *web service*. Permite el envío de mensajes de errores a través del *web service* hacia el cliente para avisar sobre cualquier flujo interrumpido de forma abrupta para poder reaccionar de forma acorde.
- HashCollider.Model.Mapping.dll: Permite identificar qué campos de la base de datos están asignadas a qué atributos de cada entidad del modelo para facilitar la persistencia en la base de datos.
- HashCollider.Manager.dll: Esta biblioteca facilita la lectura y escritura de los objetos del modelo en la base de datos.
- HashCollider.Web.dll: Es una aplicación *web* que contiene todas las páginas con contenido dinámico para que cualquier persona que acceda al sitio pueda ver en línea el estado de resolución de los fragmentos.
- HashCollider.WebService.dll: El componente contiene todo el comportamiento y coordinación requeridos en el *web service* para gestionar la fragmentación y asignación de fragmentos a los clientes de forma concurrente.
- HashCollider.WinService.dll: Representa al servicio Windows que conforma el cliente en este sistema. Este componente permitirá a equipos procesar fragmentos de forma automática y sin intervención del usuario.
- HashCollider.Winforms.dll: Es una aplicación de escritorio que permite configurar con qué frecuencia el servicio Windows descargará fragmentos. De esta forma, se puede controlar la cantidad de recursos que insumirá el cliente en el equipo. Además cuenta con un módulo de depuración que muestra todos los datos asociados al fragmento que se está resolviendo actualmente.

Diagrama de Actividad

El diagrama de actividad es un modelo para mostrar las actividades realizadas por cada rol en el sistema y cómo se interrelacionan en todo el ciclo principal del sistema.

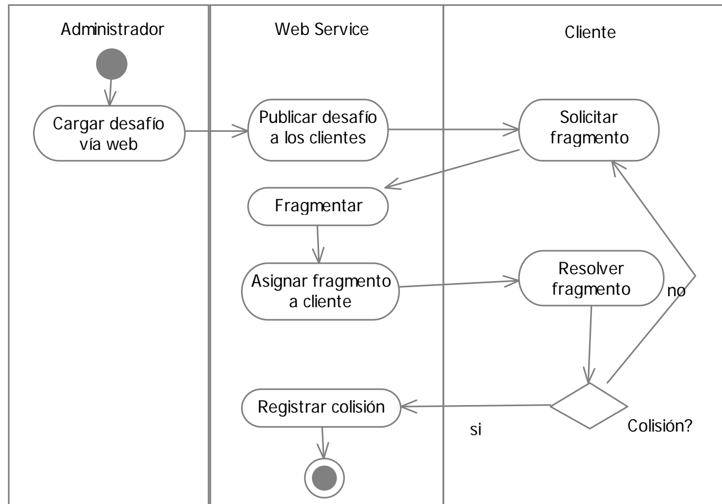


Ilustración 39 – Diagrama UML de actividad

Diagrama de Uso

Los diagramas de uso modelan la lógica y eventos realizados en un escenario dado. Se tomará de ejemplo el escenario en que un cliente solicite un fragmento y encuentre una colisión.

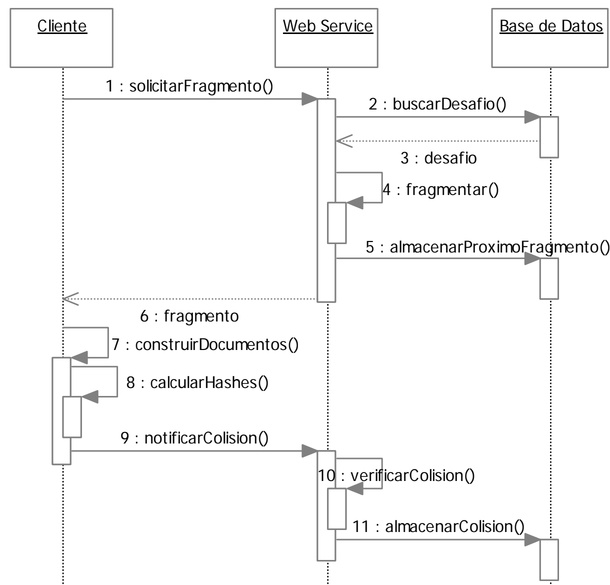


Ilustración 40 – Diagrama UML de uso

En este caso, el cliente solicita un fragmento, por lo que el *web service* le devuelve el fragmento actual y a su vez almacena el próximo fragmento para la próxima solicitud. El cliente deberá construir uno por uno los documentos a probar asignados al fragmento en cuestión. Para cada documento candidato se calculará su *hash* y se verificará si el documento genera una colisión. En caso afirmativo, el cliente notifica al *web service* el documento que genera la colisión, éste lo verifica y lo almacena.

Anexo D: Extensión de funcionalidad

El sistema ha sido diseñado basado en una creación estructurada de documentos candidatos. Estos se componen por un prefijo conocido y por un *padding*. El *padding* a su vez se compone por tres bloques: una preámbulo, el *padding per se* y un sufijo.

El motivo del preámbulo y sufijo del *padding* es para denotar el inicio y fin del *padding* respetando las reglas semánticas. Por ejemplo, en los tipos de desafíos de código fuente, los preámbulos y sufijo demarcan el inicio y fin de los comentarios utilizados para encapsular el *padding*. En el caso de documentos de texto planos y binarios, los preámbulos y sufijos están vacíos ya que carecen de sentido.

El *padding* encapsulado estará constituido únicamente por un conjunto de símbolos escogidos cuidadosamente para no generar accidentalmente el sufijo que denote el fin del *padding*. En otras palabras, en el caso de los desafíos de código fuente se evita tener dentro de los símbolos posibles a aquellos que generen el fin de los comentarios de programador según cada sintaxis.

A continuación se detallará como extender nuevos tipos de desafíos. Se asume que el lector tiene conocimientos de programación. El sistema se ha diseñado para que sea altamente adaptable y extensible.

1. Nombre y definición del nuevo desafío

Como primer paso, se deberá crear el nombre del nuevo tipo de desafío. Para realizar esto se deberá editar el código fuente de la biblioteca denominada *HashCollision.Model*, incluida dentro del software de HashCollider. Se deberá buscar la clase *ChallengeType* y *ChallengeTypeFormat* para extender el enumerado de distintos tipos de desafíos y su descripción, respectivamente.

2. Composición y comportamiento del nuevo desafío

Toda la lógica de los desafíos ha sido encapsulada en una biblioteca que se llama *HashCollision.Model* incluida dentro del software de HashCollider. La única tarea a realizar para agregar nuevos tipos de

desafíos es agregar una clase que herede de la clase abstracta *FragmentResolver* incluida en dicha biblioteca. Allí se deberán definir los siguientes métodos:

- GetPaddingPrefix(): Esta función devolverá los bytes que correspondan al preámbulo del padding. Por ejemplo, en XML el preámbulo se define como “<--”.
- GetPaddingSuffix(): Esta función devolverá los bytes que correspondan al sufijo que permitirá encapsular al *padding* y delimitar su fin. Por ejemplo, en XML el sufijo del *padding* se define como “-->”.
- GetSymbolsToTest(): Esta función devuelve la cantidad de símbolos que serán permutados en cada fragmento. Cuanto más grande sea el número, mayor será el tamaño de los fragmentos, más documentos asociados serán procesados en cada ciclo de resolución del cliente pero demandará más tiempo en ser resuelto de forma completa. Este número además permite ajustar el estado actual del desafío al desplazar el *padding* en múltiplos de este número.
- GetPermittedSymbols(): Esta función devolverá los símbolos permitidos para realizar las permutaciones y que finalmente compondrán el *padding* encapsulado.

3. Integración del nuevo tipo de desafío al software

Para que el sistema pueda integrar el nuevo tipo de desafío es necesario modificar la clase *FragmentResolverFactory* que implementa un patrón *Factory*.

4. Habilitar la selección desde el administrador de desafíos

Para que el administrador pueda generar desafíos según el nuevo tipo, es necesario incluir este nuevo tipo dentro del combo desplegable que se genera en la página *Admin/admin_challenge_upload.aspx* incluida dentro del sitio *web*.

Anexo E: Protocolo de comunicación

En este anexo se detallará la implementación y utilización de los métodos del *web service* propios de HashCollider. Estos métodos son los vectores que permiten la resolución distribuida utilizando el procesamiento en los equipos de los clientes.

Métodos publicados

El *web service* consta de tan sólo tres métodos. Estos métodos pueden ser llamados desde cualquier cliente:

- *GetChallengeFragment*: Es el primer y principal método que debe invocar el cliente. Este método será llamado para descargar todos los datos necesarios para efectuar la resolución y además el identificador del fragmento asignado. El cliente cargará dicha información en memoria y procesará todos los documentos asociados al fragmento asignado en busca de una colisión.
- *GetNextFragmentIndex*: Este método es el resultado de una mejora de desempeño de la herramienta. Para aumentar la eficiencia del motor de fragmentación y el flujo de documentos procesados, este método deberá ser llamado luego de *GetChallengeFragment* y de forma indefinida o hasta que HashCollider informe lo contrario.

El método es sencillo e implica un tiempo de respuesta muy por debajo del primer método. Una vez obtenido el identificador del fragmento asignado, al invocar este método el cliente obtendrá el identificador del próximo fragmento asignado. Sin la necesidad de descargar todos los datos de nuevo ya que el cliente los tiene residentes en memoria, se puede realizar un desplazamiento rápido y eficiente hacia el próximo fragmento asignado. Durante este desplazamiento, el cliente posiciona el sufijo secuencial según el índice del próximo fragmento a resolver. De esta manera, se evita la transferencia reiterada de los datos del fragmento.

- *NotifyCollision*: Este método se invoca únicamente cuando un cliente ha hallado una colisión. Se le envía al *web service* el documento que genera la colisión y luego de ser verificado se lo registra en el sistema. El

hallazgo de una colisión disparará un email automático a los suscriptos en la lista de distribución.

A continuación se tabularon los detalles sobre los distintos métodos publicados:

Método	Entrada	Salida	¿Cuándo de invoca?
GetChallengeFragment	N/A	- Datos del desafío - Sufijo secuencial - Identificador del fragmento asignado	- Para resolver un fragmento sin tener los datos del fragmento en memoria. - Se llama por primera vez para inicializar el cliente.
GetNextFragmentIndex	- Identificador de desafío	- Identificador del próximo fragmento asignado	- Para resolver un fragmento ya teniendo los datos en memoria. - Se debe llamar después de *GetChallengeFragment*.
NotifyCollision	- Identificador de desafío - Documento que genera la colisión	N/A	- Al hallar un documento que genere la colisión.

Tabla 41 – Métodos del *web service*

Protocolo de comunicación

La serie de pasos que debe efectuar un cliente para interactuar con el *web service* se lista a continuación:

1. Llamar a *GetChallengeFragment*. El cliente obtendrá el identificador del fragmento asignado y todos los datos relacionados al desafío (por ejemplo, el identificador de desafío, función de *hashing*, tipo de desafío, etc). El cliente cargará estos datos en memoria y procederá a recorrer todos los documentos asignados al fragmento.
2. En caso de hallar una colisión, se invocará a *NotifyCollision*. De esta forma la colisión será registrada en el sistema. Al finalizar, se purgarán los datos almacenados en memoria obligando a ir al Paso 1.
3. Si no se encuentra una colisión, el cliente llamará a *GetNextFragmentId*. El resultado de dicho método será el identificador del próximo fragmento asignado. El cliente ya tiene toda la información almacenada en memoria y sabe cuál fue el identificador de fragmento recientemente procesado. Conociendo el próximo identificador de fragmento asociado, el cliente realizará un desplazamiento lógico en memoria para

posicionarse en el nuevo fragmento asignado. Se procede a recorrer nuevamente los documentos asignados en busca de una colisión.

4. En caso de hallar una colisión, ir al Paso 2.
5. En caso de no hallar una colisión, ir al Paso 3.

El flujo de invocaciones a los métodos del *web service* se grafica a continuación:

Ilustración 42 – Diagrama de flujo del protocolo de comunicación

Anexo F: Acceso *web* a HashCollider

HashCollider incluye un portal *web* que permite publicar el estado de todas las colisiones en tiempo real. Cualquier persona en el mundo podrá acceder al avance del proyecto, registrar su correo electrónico para recibir noticias, descargar software y contribuir con la resolución de los desafíos publicados. Además, el administrador del sitio podrá mantener el listado de desafíos publicados.

Página de inicio

En la página de inicio del sitio, a nivel de resumen, se detallará la cantidad de colisiones halladas, la cantidad total de documentos procesados, la cantidad de gente suscripta a los emails y el total de descargas. Además, se listan los distintos tipos de desafíos disponibles en HashCollider: bash, binario, C/C++, Haskell, HTML, Java, Perl, PHP, Python, Ruby, texto, Visual Basic y XML.

Ilustración 43 – Página de inicio

El portal es fácilmente navegable a través de un menú en la sección de arriba. Utilizando este menú, el usuario podrá:

- Acceder al estado de cada desafío publicado.

- Analizar las estrategias de resolución de colisiones para cada tipo de desafío debido a las reglas semánticas implícitas.
- Descargar el software para instalar un cliente y colaborar con la resolución de desafíos.
- Entender cómo funciona HashCollider.
- Registrar su correo electrónico a la lista de distribución.
- Ingresar al panel de administración.

Avance en la resolución de los desafíos

El usuario podrá acceder el listado de desafíos publicados, visualizar su descripción, el tipo de desafío, la función de *hashing* empleada y su fecha de creación. A su vez, podrán descargar el archivo original y el contenido deseado con el que se desea fraguar el documento.

Ilustración 44 – Página de estado de desafíos

Los resultados en la grilla se actualizan de forma periódica y automática para que los usuarios puedan ver de forma interactiva cómo aumenta la cantidad de documentos procesados. La frecuencia de refresco es configurable por el usuario.

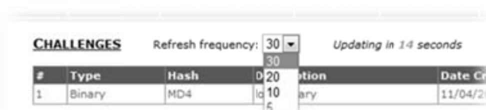


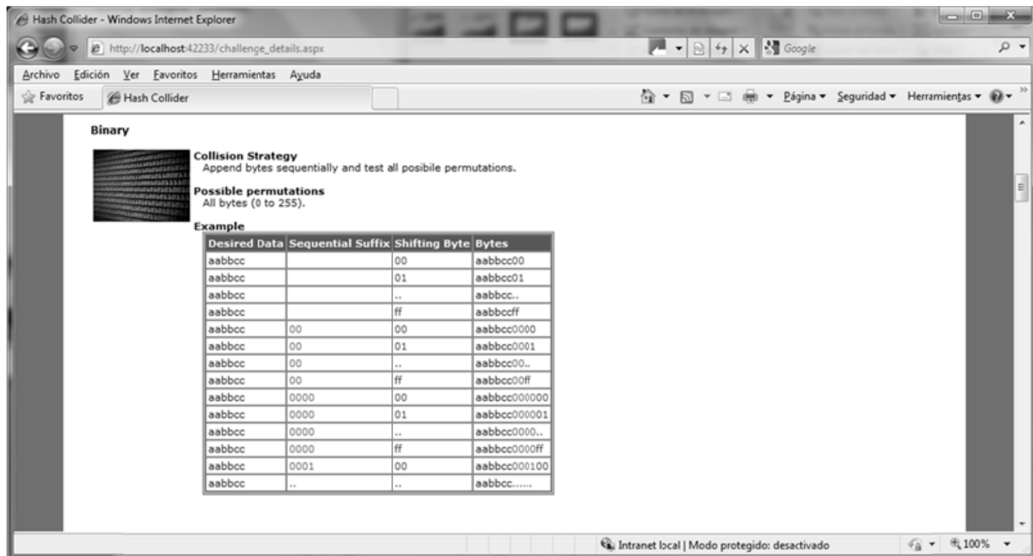
Ilustración 45 – Configuración de frecuencia de actualización de estado

Estrategias de resolución

El usuario podrá escudriñar en las distintas estrategias aplicadas para la construcción de un *padding* que respete las reglas semánticas del documento y a su vez generar la colisión. Se podrá optar entre los distintos tipos de desafío y ver su estrategia asociada.

Ilustración 46 – Página de estrategias de resolución

Para cada uno de los tipos de desafíos, se detalla brevemente cómo generar el *padding* y se brinda un ejemplo práctico para una mejor comprensión.



Desired Data	Sequential Suffix	Shifting Byte	Bytes
aabbcc		00	aabbcc00
aabbcc		01	aabbcc01
aabbcc		..	aabbcc..
aabbcc		ff	aabbccff
aabbcc	00	00	aabbcc0000
aabbcc	00	01	aabbcc0001
aabbcc	00	..	aabbcc00..
aabbcc	00	ff	aabbcc00ff
aabbcc	0000	00	aabbcc000000
aabbcc	0000	01	aabbcc000001
aabbcc	0000	..	aabbcc0000..
aabbcc	0000	ff	aabbcc0000ff
aabbcc	0001	00	aabbcc000100
aabbcc	..	..	aabbcc......

Ilustración 47 – Página con estrategia de resolución para archivos binarios

Descarga del *software* cliente

El sitio *web* contiene una página donde cualquier persona en el mundo podrá descargar e instalar el cliente de HashCollider para colaborar con la resolución de desafíos. Una vez instalado, el software puede ser configurado para determinar con qué frecuencia se estarán resolviendo fragmentos.

Ilustración 48 – Página de descarga de la herramienta

Actualmente, tal como se ha mencionado anteriormente, los clientes están desarrollados para la plataforma Windows. Sin embargo, debido al diseño y a la utilización de *web services*, es posible la construcción de nuevos clientes para distintas plataformas. El único requerimiento para desarrollar estos nuevos clientes es consumir el *web service* e implementar su protocolo de resolución de fragmentos.

Resumen de aspectos teóricos

Los usuarios podrán acceder a una página que describe de forma resumida y concisa los aspectos teóricos sobre las colisiones de *hash*. De esta forma, una persona que no comprende en profundidad el fundamento sobre el cual opera HashCollider, podrá absorber las ideas principales. En esta página se explica qué es una función de *hashing*, para qué se utiliza, qué es una firma digital, por qué existen colisiones, cómo generar colisiones respetando las reglas semánticas del tipo de desafío y qué implicancias tendrá la resolución de un desafío.

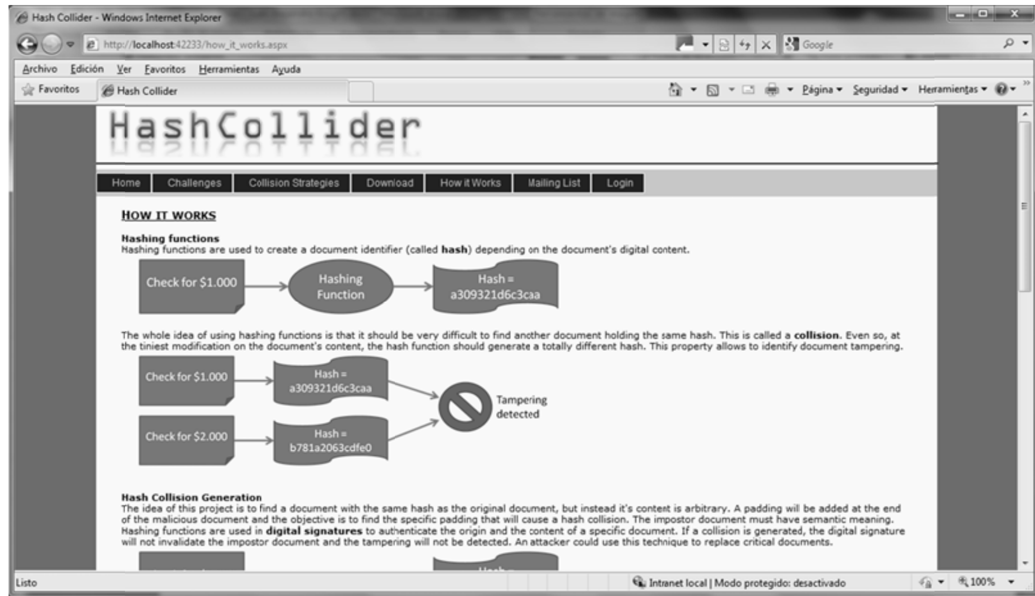


Ilustración 49 – Página de explicación de la herramienta

Notificaciones vía email

Cualquier interesado podrá registrar su correo electrónico en la lista de distribución de noticias. Una vez registrado, se podrá enterar de las mejoras en el sistema, avances en los desafíos y nuevas descargas de software. Principalmente, la razón más atractiva de la lista de correos es la notificación automática al hallarse una colisión.

Como toda lista de distribución de correos electrónicos, también existe la posibilidad de anular la suscripción para no ser categorizados como correo no solicitado (SPAM).

Para poder enviar los correos de forma automática, se ha creado una cuenta en Yahoo!: hashcollider@yahoo.com.ar.

Ilustración 50 – Página de suscripción a la lista de distribución

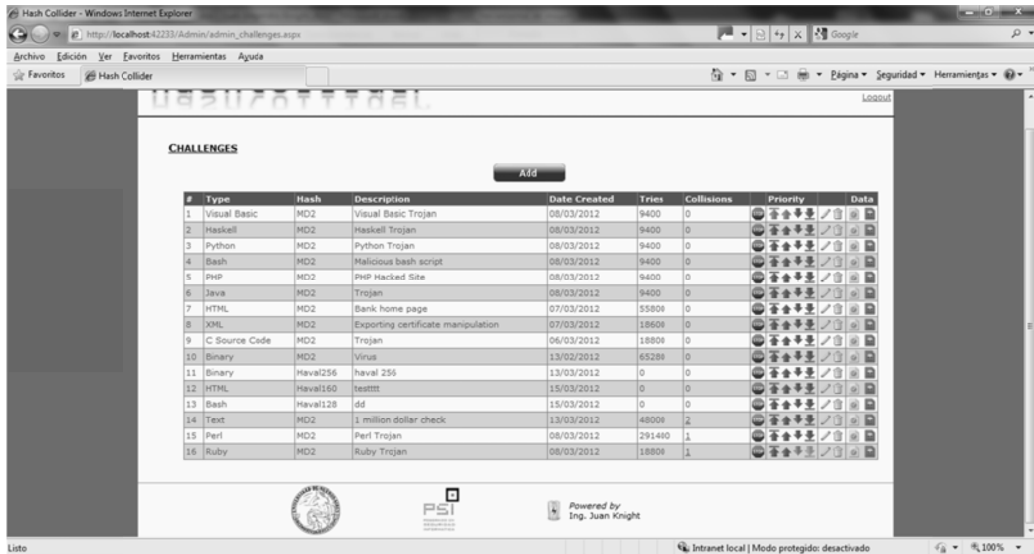
Acceso al panel de administración

El portal *web* contiene un panel de administración de los desafíos del sistema. Para poder acceder a dicha funcionalidad, el administrador deberá ser autenticado e ingresar su usuario y contraseña.

Ilustración 51 – Página de autenticación

Administración de desafíos

El administrador podrá gestionar los desafíos cargados en el sistema y acceder a toda la información vinculada.



#	Type	Hash	Description	Date Created	Tries	Collisions	Priority	Data
1	Visual Basic	MD2	Visual Basic Trojan	08/03/2012	9400	0		
2	Haskell	MD2	Haskell Trojan	08/03/2012	9400	0		
3	Python	MD2	Python Trojan	08/03/2012	9400	0		
4	Bash	MD2	Malicious bash script	08/03/2012	9400	0		
5	PHP	MD2	PHP Hacked Site	08/03/2012	9400	0		
6	Java	MD2	Trojan	08/03/2012	9400	0		
7	HTML	MD2	Bank home page	07/03/2012	55800	0		
8	XML	MD2	Exporting certificate manipulation	07/03/2012	18600	0		
9	C Source Code	MD2	Trojan	06/03/2012	18800	0		
10	Binary	MD2	Virus	13/02/2012	65280	0		
11	Binary	Haval256	haval 256	13/03/2012	0	0		
12	HTML	Haval160	testttt	15/03/2012	0	0		
13	Bash	Haval128	dd	15/03/2012	0	0		
14	Text	MD2	1 million dollar check	13/03/2012	48000	2		
15	Perl	MD2	Perl Trojan	08/03/2012	291400	1		
16	Ruby	MD2	Ruby Trojan	08/03/2012	18800	1		

Ilustración 52 – Página de administración de desafíos

Podrá visualizar los detalles de los desafíos, la cantidad de documentos ya procesados y la cantidad de colisiones halladas. Además, el administrador podrá descargar todos los documentos asociados a un desafío, es decir, el documento original, el documento con el contenido que se desea colisionar y los documentos que generan dicha colisión si es que fueron hallados.

Una premisa básica en la forma de operar en HashCollider es que el desafío a ser fragmentado y resuelto es el que tenga la menor prioridad. Para poder cambiar las prioridades de los desafíos, el administrador podrá reordenarlos a su gusto utilizando las flechas en la botonera de la grilla. Las opciones de reordenamiento son: establecer como el primer desafío con menor prioridad, bajar una posición, subir una posición o establecer como el último desafío de mayor prioridad.

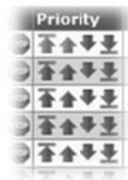


Ilustración 53 – Botonera para cambiar las prioridades de desafíos

Crear un nuevo desafío

Para poder ingresar un nuevo desafío, el administrador deberá asignar el tipo de desafío, la función de *hashing* a implementar, la descripción del motivo de resolución del desafío, el archivo original y el archivo deseado que suplantará al original. Al definir estos datos, el sistema toma el archivo original y aplica la función de *hashing* escogida, por lo ya habrá calculado el *hash* objetivo de la colisión. Tomando la información recién ingresada, HashCollider creará documentos candidatos según el tipo de documento y buscará una colisión.

Ilustración 54 – Página de alta de desafío

El administrador también puede configurar la naturaleza de la resolución del desafío. Si elige detener la resolución cuando se encuentre la primera colisión, en caso de hallarla, el desafío es reasignado con la prioridad más baja, por lo que el desafío con menor prioridad será el próximo en ser resuelto. Esto permite un grado de automatización en la resolución de varios desafíos y no depende de la interacción de un administrador para ir rotando de forma manual la resolución de desafíos.

Si el administrador elige no detener la resolución al hallar una colisión, sencillamente se seguirá resolviendo el mismo desafío buscando la próxima colisión. En caso de querer cambiar de desafío que actualmente se está resolviendo, el administrador simplemente debe asignar a otro desafío la prioridad más baja.

Editar un desafío existente

Una vez creado un desafío, muchos de sus datos no podrán ser modificados. El administrador podrá acceder a la edición de un desafío utilizando la grilla y únicamente podrá modificar su descripción y determinar si su resolución culmina al encontrar la primera colisión o si continuará buscando otra colisión.

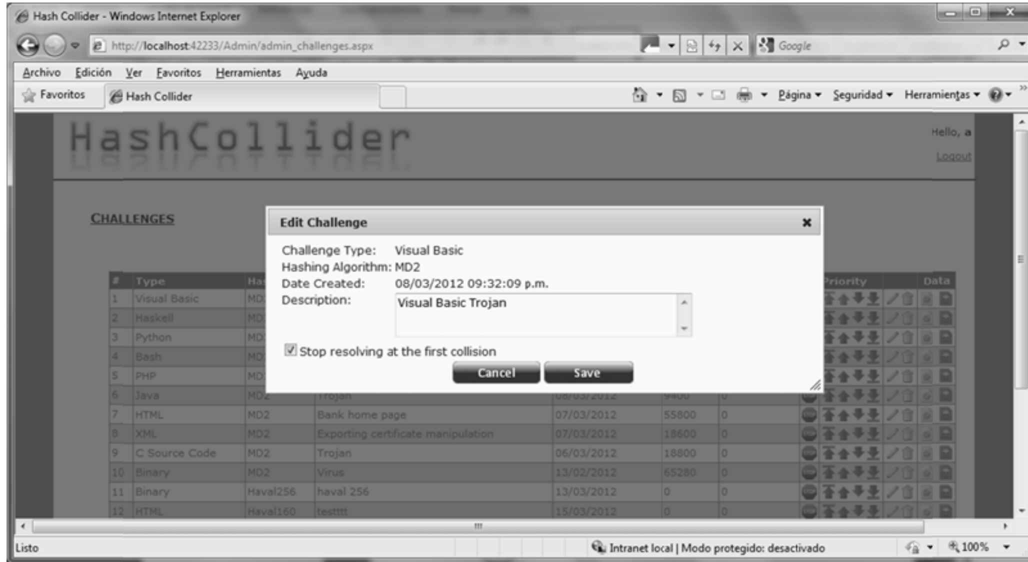


Ilustración 55 – Página de edición de desafío

Fuentes

Las fuentes que dieron el origen a las referencias bibliográficas fueron efectuadas en su gran mayoría mediante el buscador *web* de Google. De forma complementaria, los tutores del presente trabajo recomendaron al libro escrito por Menezes para brindar solidez a la teoría e implementación de la criptografía.

Bibliografía

[1] Alfred Menezes, Paul van Oorschot y Scott Vanstone, Handbook of Applied Cryptography, CRC Press, Octubre de 1996

[2] XML
http://www.w3.org/XML (consultada el 5/7/2012)

[3] SHA1
http://tools.ietf.org/html/rfc3174 (consultada el 5/7/2012)

[4] MD5
http://www.ietf.org/rfc/rfc1321.txt (consultada el 5/7/2012)

[5] HMAC
http://www.ietf.org/rfc/rfc2104.txt (consultada el 5/7/2012)

[6] Digital Watermarking
http://www.digitalwatermarkingalliance.org (consultada el 5/7/2012)

[7] Microsoft .Net Framework
http://www.microsoft.com/net (consultada el 06/07/2012)

[8] SQL Injection
https://www.owasp.org/index.php/SQL_Injection (consultada el 5/7/2012)

[9] X509 Certificates
http://tools.ietf.org/html/rfc3280 (consultada el 5/7/2012)

[10] Cain & Abel
http://www.oxid.it/cain.html (consultada el 5/7/2012)

[11] John The Ripper
http://www.openwall.com/john/ (consultada el 5/7/2012)

[12] l0phtcrack
http://www.l0phtcrack.com/ (consultada el 5/7/2012)

[13] Rainbow Crack Project
http://project-rainbowcrack.com (consultada el 30/11/2011)

[14] Marc Stevens, Arjen Lenstra y Benne de Weger, Chosen-prefix Collisions for MD5 and Applications, 27/06/2011
http://www.win.tue.nl/hashclash/EC07v2.0.pdf (consultada el 30/11/2011)

[15] Marc Stevens, Arjen Lenstra y Benne de Weger, 30/11/2007
http://www.win.tue.nl/hashclash/SoftIntCodeSign/ (consultada el 30/11/2011)

[16] Arjen Lenstra, Xiaoyun Wang y Benne de Weger, Colliding X.509 Certificates, 01/03/2005
http://www.win.tue.nl/~bdeweger/CollidingCertificates/CollidingCertificates.pdf
(consultada el 30/11/2011)

[17] Analyzing the MD5 collision in Flame
http://trailofbits.files.wordpress.com/2012/06/flame-md5.pdf (consultada el 16/06/2012)

[18] ACryptoHashNet Project
http://sourceforge.net/projects/acryptohashnet/ (consultada el 30/11/2011)

[19] Buffer Caché
http://www.research.ibm.com/people/d/dfb/papers/Bacon94Cache.pdf
(consultada el 06/07/2012)

[20] Aceleración algorítmica
http://www.cs.cf.ac.uk/Parallel/Year2/section7.html (consultada el 09/07/2012)

[21] Windows 7 Home Premium
http://windows.microsoft.com/en-us/windows7/products/home (consultada el 09/07/2012)

[22] Intel Core i5
http://www.intel.com/content/www/us/en/processors/core/core-i5-processor.html
(consultada el 09/07/2012)

[23] Visual Studio 2010
http://www.microsoft.com/visualstudio/en-us/products/2010-editions
(consultada el 09/07/2012)

[24] Web service
http://www.w3.org/TR/ws-arch/ (consultada el 09/07/2012)

[25] HTTP
http://www.w3.org/Protocols/rfc2616/rfc2616.html (consultada el 09/07/2012)

[26] SOAP
http://www.w3.org/TR/soap/ (consultada el 09/07/2012)

[27] UML
http://www.uml.org/ (consultada el 25/07/2012)

[28] Marc Stevens, Master's Thesis On Collisions for MD5, Junio 2007
http://www.win.tue.nl/hashclash/On%20Collisions%20for%20MD5%20-%20M.M.J.%20Stevens.pdf (consultada el 25/07/2012)

Bibliografía General

Para una mejor comprensión del presente trabajo sería provechoso que el lector pueda leer material y comprender los siguientes temas:

- Seguridad Informática
- Funciones de *hashing* criptográficos
- Firmas digitales
- Criptoanálisis de las funciones de *hashing*

El libro citado escrito por Menezes trata todos estos temas de una forma clara y didáctica.

Índice de Ilustraciones y Tablas

Ilustración 1 - Función de *Hashing* 14
Ilustración 2 - Conjuntos de una función de *hashing* 16
Ilustración 3 - Conjuntos en una colisión de *hash* 16
Ilustración 4 – Detección de documentos adulterados comparando los *hashes* 17
Ilustración 5 – Resistencia de pre-imagen 18
Ilustración 6 – Segunda resistencia a la pre-imagen 19
Ilustración 7 – Resistencia de colisiones 20
Ilustración 8 – Analogía entre firma hológrafa y digital 20
Ilustración 9 – Firma digital 21
Tabla 10 – Comparación entre firma hológrafa y digital 22
Ilustración 11 – Firma digital de documento usando un certificado X509 22
Ilustración 12 – Colisión con documento ilegible 23
Ilustración 13 – Colisión con prefijo fijo 23
Ilustración 14 – Comparación de estrategias de colisón 24
Ilustración 15 – Países afectados por ataque Flame a MD5 (Fuente: Kaspersky Labs) 26
Ilustración 16 – Contenido de un desafío 27
Tabla 17 – Familias de tipos de documento 28
Tabla 18 – Tipos de Desafíos 28
Ilustración 19 – Segmentación de desafío en fragmentos 29
Ilustración 20 – Composición y armado de documento candidato 29
Ilustración 21 – Contenido de un fragmento 30
Ilustración 22 – Motor de fragmentación 31
Ilustración 23 – Arquitectura de HashCollider 31
Ilustración 24 – Plataforma de clientes para el web service 33
Ilustración 25 – Fragmentación correlativa 34
Ilustración 26 – Documentos procesados en un fragmento 35
Tabla 27 – Estrategias de armado de documentos candidatos 36
Ilustración 28 – Sintáxis de comentarios para distintos lenguajes de programación 37
Ilustración 29 – Buffer Caché 37
Ilustración 30 – Transmisión completa de fragmentos 39
Ilustración 31 – Transmisión y desplazamiento automático de fragmentos 40
Ilustración 32 – Diseño de archivos de prueba 40
Ilustración 33 – Pruebas realizadas 41
Tabla 34 – Duración de un ataque a MD5 según cantidad de nodos 44
Tabla 35 – Duración de un ataque a MD5 utilizando el enfoque de Stevens 45
Ilustración 36 – Diagrama UML de despliegue 53
Ilustración 37 – Diagrama UML de procesos 54
Ilustración 38 – Diagrama UML de componentes 54
Ilustración 39 – Diagrama UML de actividad 56
Ilustración 40 – Diagrama UML de uso 56
Tabla 41 – Métodos del *web service* 61
Ilustración 42 – Diagrama de flujo del protocolo de comunicación 62

Ilustración 43 – Página de inicio 63
Ilustración 44 – Página de estado de desafíos........ 64
Ilustración 45 – Configuración de frecuencia de actualización de estado 64
Ilustración 46 – Página de estrategias de resolución 65
Ilustración 47 – Página con estrategia de resolución para archivos binarios........ 65
Ilustración 48 – Página de descarga de la herramienta........ 66
Ilustración 49 – Página de explicación de la herramienta 67
Ilustración 50 – Página de suscripción a la lista de distribución........ 68
Ilustración 51 – Página de autenticación........ 68
Ilustración 52 – Página de administración de desafíos........ 69
Ilustración 53 – Botonera para cambiar las prioridades de desafíos 69
Ilustración 54 – Página de alta de desafío 70
Ilustración 55 – Página de edición de desafío........ 71