Universidad Internacional de La Rioja

Escuela Superior de Ingeniería y Tecnología

Máster Universitario en Seguridad Informática

Desarrollo de un software educativo para la función hash BLAKE2

Trabajo fin de estudio presentado por:	Ismael Vélez Recio
Tipo de trabajo:	Desarrollo software
Director/a:	Rodolfo García Peñas
Fecha:	21/07/2022

Resumen

En la actualidad existen multitud de algoritmos y mecanismos para la criptografía. Estos son cada vez más complejos, lo cual dificulta su aprendizaje por parte de los alumnos.

Aunque ya existen diversos programas de apoyo para los alumnos (AESphere, Flujolab, safeDES, CrypTool, etc.), estos se podrían abordar con un enfoque diferente para facilitar su comprensión. Además, todavía hay algoritmos para los cuales no existe una herramienta similar a las mencionadas o esta no muestra en detalle los cálculos internos del algoritmo.

Por ello, se pretende realizar un análisis del estado del arte, para posteriormente realizar el diseño, desarrollo y validación de una herramienta que permita a los alumnos comprender estos conceptos de una forma más sencilla.

El algoritmo para el que se ha realizado el software es la función de hash BLAKE2.

Palabras clave: Criptografía, educativo, software, BLAKE2.

Abstract

Nowadays there are a multitude of algorithms and mechanisms for cryptography. These are becoming increasingly complex, which makes it difficult for students to learn them.

Although there are already several support programs for students (AESphere, Flujolab, safeDES, CrypTool, etc.), these could be addressed with a different approach to enhance their understanding. Moreover, there are still algorithms for which a tool similar to those mentioned does not exist or does not show in detail the internal computations of the algorithm.

Therefore, it is intended to perform an analysis of the state of the art, and then design, develop and validate a tool that allows students to understand these concepts in a simpler way.

The algorithm for which the software has been developed is the BLAKE2 hash function.

Keywords: Cryptography, educational, software, BLAKE2.

Índice de contenidos

Índice de figuras

Índice de tablas

1. Introducción

Cuando la informática estaba en sus inicios, la seguridad de la información y las comunicaciones no eran factores que habitualmente se tuvieran en cuenta. Sin embargo, con el crecimiento y evolución de Internet esto ha cambiado considerablemente y en la actualidad tanto los usuarios de a pie como las grandes empresas están más concienciados sobre ello, aunque todavía quede un gran camino por recorrer.

Uno de los principales elementos que ha permitido el desarrollo de la seguridad informática es la criptografía, ya que esta nos permite poder almacenar o transmitir la información utilizando ordenadores de forma segura (Gençoğlu, 2019).

La criptografía consiste en transformar la información utilizando algoritmos matemáticos a un estado ilegible, de forma que podamos garantizar que solamente pueda recuperar esa información el destinatario legítimo. La seguridad de esta transformación radica en que, aunque es posible recuperar la información sin tener la clave, requiere de una capacidad de cómputo y tiempo tan grande que en la práctica es imposible de realizar.

Sin embargo, también existen métodos criptográficos irreversibles, como sería el caso del hashing. Estos algoritmos transforman cualquier tipo de datos en una cadena de texto de tamaño fija y, debido a los mecanismos que utilizan para generarla, no permiten recuperar el contenido original.

La criptografía no es algo que surgió con la computación. El sistema más antiguo que se conoce es la escítala, un bastón en el cual se enrollaba una cinta en la que se escribía el mensaje. Al desenrollar la cinta, el mensaje ya no tenía relación con el original. Es un mecanismo muy básico pero que permite entender a grandes rasgos cómo funciona la criptografía.

A partir de este momento, los diferentes métodos criptográficos han ido evolucionando y haciéndose cada vez más complejos y elaborados, lo cual a su vez dificulta su estudio a las personas que intentan aprender y entender su funcionamiento. Este es el foco de atención del presente trabajo para el desarrollo de un software educativo de criptografía.

1.1. Motivación

Existen una gran variedad de enfoques y tipos de algoritmos criptográficos dependiendo de las necesidades del problema que se quiera resolver y las capacidades de la tecnología existente en el momento en el que estos son desarrollados.

Además, la creciente complejidad de los algoritmos que se utilizan en criptografía hace que su estudio por parte de la gente que está adentrándose en el mundo de la criptografía moderna sea mucho más complicado, sobre todo si no se tiene una buena base de conocimientos sobre matemáticas.

En los últimos años se han desarrollado muchas herramientas que facilitan esto (como se verá en el capítulo 2.3), sin embargo, muchas de estas herramientas se limitan a mostrar el resultado final o, como mucho, mostrar alguno de los cálculos intermedios.

Para algunos algoritmos (como Keccak, DES o AES), debido a que son los más utilizados y han pasado muchos años de su creación, existen herramientas muy completas que permite entender fácilmente el funcionamiento paso a paso de estos algoritmos. No obstante, hay otros algoritmos, también muy usados como BLAKE2, para los cuales estas herramientas no existen.

Debido a esto, creo que el desarrollo de un software educativo para este algoritmo sería muy beneficioso como medio complementario de estudio para todos aquellos que buscan comprender estos conceptos tan complejos.

1.2. Planteamiento del trabajo

Como ya se ha comentado, uno de los principales problemas de los algoritmos más recientes es que son bastante complejos y no hay herramientas educativas de ellos, lo cual dificulta su estudio.

Es debido a esto que surgió la idea de desarrollar una herramienta educativa, en este caso para BLAKE2. El objetivo es, realizando un estudio de las herramientas existentes para otros algoritmos, realizar el diseño de un software que permita solventar esta carencia, de forma que se facilite el estudio de este algoritmo al mostrar cómo funciona de la forma más clara y sencilla posible.

1.3. Estructura del trabajo

Capítulo 2. Estado del arte

En este capítulo se realiza un estudio del estado del arte, analizando y comparando diversos estudios, trabajos, prototipos y sistemas actuales que abordan una problemática próxima a la del presente trabajo, poniendo el foco en las diferentes herramientas ya existentes en dichas propuestas, y los resultados obtenidos.

Además, se comentará la elección final del algoritmo y se realizará una explicación detallada de la estructura y funcionamiento del mismo.

Capítulo 3. Objetivos

En este capítulo se procederá a determinar los objetivos que persigue el desarrollo del software educativo a través de este TFM. Se determinará el objetivo general y los objetivos específicos que, en su conjunto, permitirán lograr el objetivo general.

Por último, se explicará la metodología de trabajo, el marco tecnológico y la planificación del desarrollo del software.

Capítulo 4. Análisis y diseño de la aplicación

En este capítulo se especificarán los requisitos funcionales y no funcionales junto a una breve descripción de los mismos con el objetivo de identificar las necesidades que el software debe satisfacer para cumplir con los objetivos definidos en el capítulo 3. Además, se incluirán los casos de uso, bocetos y diagramas iniciales.

Capítulo 5. Resultados

En este capítulo se exponen los resultados obtenidos del desarrollo del presente TFM.

Esto incluye el proceso de desarrollo, la arquitectura del sistema, el aspecto visual final del mismo, testeo y problemas surgidos durante el desarrollo.

Capítulo 6. Conclusiones

En este capítulo se comentarán las conclusiones obtenidas tras el desarrollo del TFG, los objetivos que se han alcanzado y las posibles mejoras a realizar en sus próximas versiones.

Anexo A. Código fuente

En este anexo se indica dónde está almacenado el código fuente del software desarrollado.

Anexo B. Guía de instalación

En este anexo se describen los pasos para realizar la instalación del software desarrollado.

2. Estado del arte

Previo al desarrollo de este proyecto se ha realizado un extenso estudio de los conceptos, desarrollos y tecnologías que constituyen su base de conocimiento. Para ello, en este apartado se analizan y comparan diversos trabajos, prototipos y sistemas actuales que abordan una problemática próxima a la del presente Trabajo de Fin de Master (TFM), poniendo el foco en las diferentes herramientas ya existentes en dichas propuestas, y los resultados obtenidos.

2.1. Contexto histórico sobre la criptografía

Como ya se ha comentado, la criptografía nos permite proteger información utilizando algoritmos matemáticos, los cuales pueden ser de distintos tipos. La primera clasificación sería la de sistema de criptografía clásica y criptografía moderna.

En criptografía, existen los llamados *principios de Kerckhoffs*, que describen las características más deseables de un sistema criptográfico:

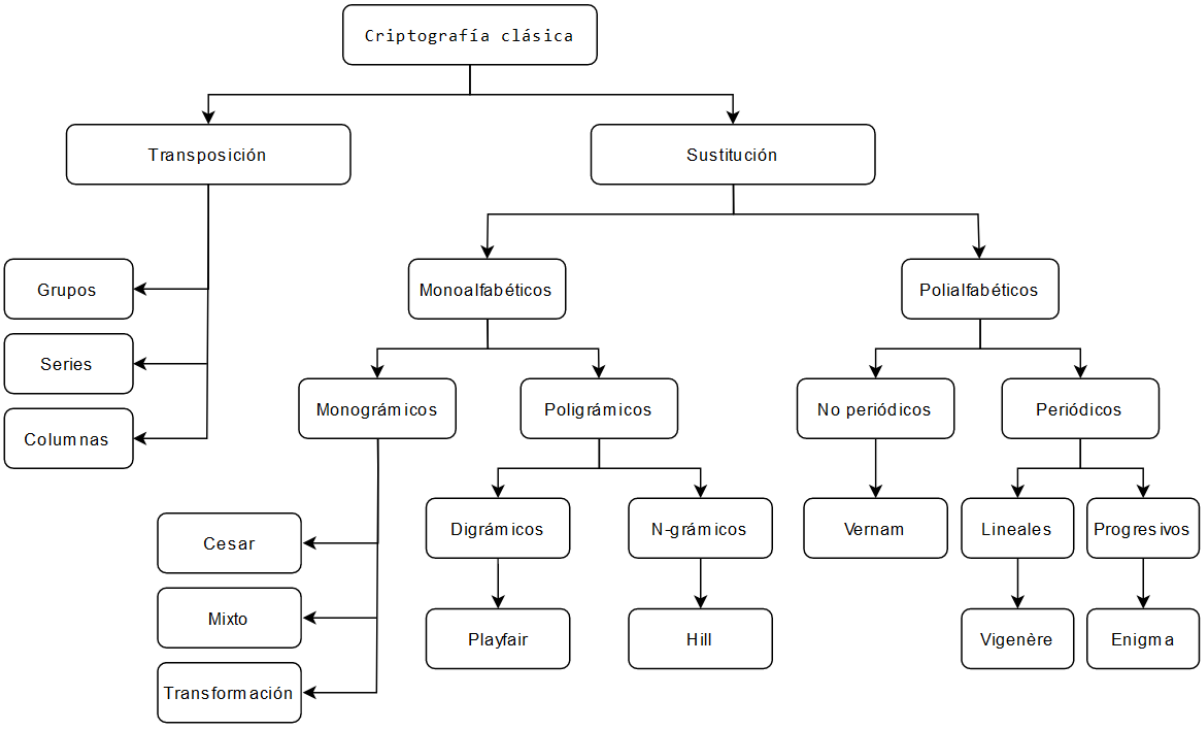
- Debe ser irrompible, al menos en la práctica.
- Su efectividad no debe depender de que su diseño sea secreto.
- La clave debe ser fácilmente memorizable.
- Los criptogramas deberán dar resultados alfanuméricos.
- El sistema debe ser operable por una única persona.
- El sistema debe ser fácil de utilizar.

La más relevante es la segunda, muy importante en la actualidad, ya que los diseños de algoritmos actuales casi siempre son públicos. Otra forma de entender esta frase es decir que la seguridad del sistema debe recaer únicamente en la clave.

2.1.1. Criptografía clásica

La criptografía clásica es muy básica y opera únicamente con las letras del alfabeto. Son bastante sencillos de descifrar sin la clave (mediante fuerza bruta, análisis de frecuencias, el método Kasiski, etc.), por lo que no se usan en la actualidad. Algunos ejemplos son la escítala (cifrado por transposición), el cifrado del César (cifrado por sustitución monoalfabética monográmica), cifrado por matrices de Hill (cifrado por sustitución monoalfabética poligrámica) o el cifrado de Vigenére (cifrado por sustitución polialfabética). La clasificación completa y algunos ejemplos de algoritmos se pueden ver en la **figura 1**.

Figura 1. Clasificación de los métodos de cifra clásicos.

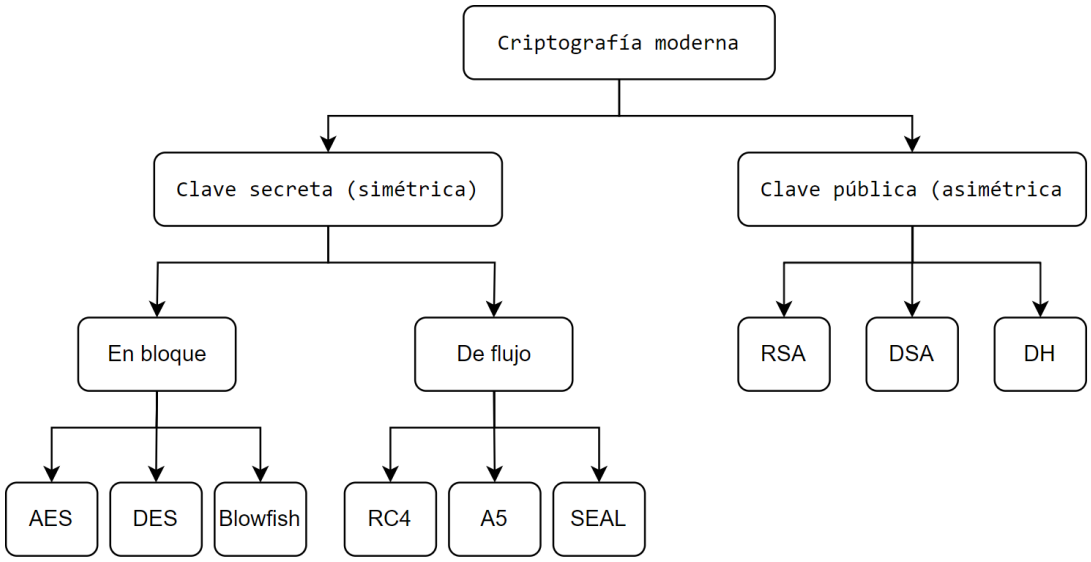


Fuente: Elaboración propia.

2.1.2. Criptografía moderna

Por otro lado, tenemos la criptografía moderna, en la cual el cifrado se realiza sobre bits o bytes en lugar directamente sobre las letras. La clasificación se puede ver en la **figura 2**.

Figura 2. Clasificación de los métodos de cifra modernos.



Fuente: Elaboración propia.

Existen dos tipos de métodos criptográficos modernos atendiendo a la cantidad de claves usadas: cifrado asimétrico o de clave pública (utiliza dos claves) y cifrado simétrico o de clave

secreta (utiliza una clave). Sus características principales, como son sus usos, velocidad y características de las claves, se pueden ver y comparar en la **tabla 1**.

Tabla 1. ***Comparativa cifrado simétrico y asimétrico.***

	Cifrado simétrico	**Cifrado asimétrico**
Tiempo en uso	Miles de años	Menos de 50 años
Ejemplos	DES, 3DES, IDEA, AES, Blowfish	DH, RSA, DSA, ElGamal
Usos	Confidencialidad, cifrado de mensajes	Confidencialidad, cifrado de mensajes, firma digital, intercambio de claves
Velocidad	Rápido (MB/s)	Lento (KB/s)
Diferencias en la clave	Número elevado (mala gestión) Vida corta (sesión)	Número reducido (buena gestión) Vida larga (años)
Tamaños de clave[1]	Mínimo 128 bits	Mínimo 1024 bits
Intercambio de claves	Difícil por un canal inseguro	La pública por cualquier canal. La privada no se comparte

Fuente: Elaboración propia.

La diferencia en la cantidad de claves es muy importante. En el asimétrico, al tener una única clave, si esta es comprometida durante, por ejemplo, el intercambio (ya que es este proceso es necesario), todos los documentos que ya hayan sido cifrados podrían ser revelados a un tercero.

Sin embargo, con el asimétrico tenemos una clave a la que puede tener acceso todo el mundo (la pública), y otra que solo nos pertenece a nosotros (la privada). Al no tener que realizar un intercambio de secretos, se reduce la probabilidad de tener un problema de seguridad.

También hay que tener en cuenta otro factor a la hora de elegir uno u otro: la velocidad. Mientras que la encriptación simétrica permite cifrar a megabytes/s, la asimétrica suele cifrar a kilobytes/s. Es decir, la simétrica es unas mil veces más rápida. Este es un factor que se

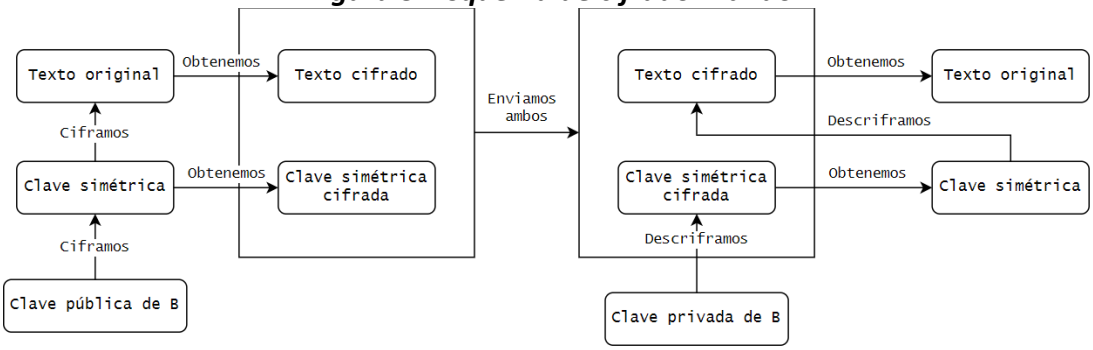
[1] Recomendaciones del NIST

vuelve muy relevante a la hora de cifrar grandes cantidades de datos, pero que se puede mitigar como veremos a continuación.

En la actualidad ambas son utilizadas, normalmente a la vez. La criptografía asimétrica se usaría para el intercambio de clave de sesión y la autenticación mediante firma digital, y la criptografía simétrica para el cifrado propiamente dicho de los datos.

Esto se hace así debido a que de esta forma se aprovechan más las fortalezas de cada una. Por un lado, tenemos la conveniencia del sistema asimétrico respecto al intercambio de claves, y por otro lado la velocidad de la simétrica. En la **figura 3** se puede observar los pasos que se siguen durante el cifrado híbrido.

Figura 3. Esquema de cifrado híbrido.

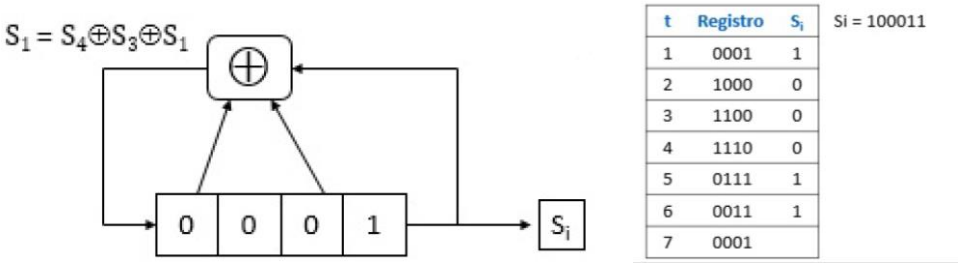


Fuente: Elaboración propia.

Volviendo a la **figura 2**, se puede ver que los algoritmos de cifrado simétrico se pueden dividir en dos categorías: en flujo o en bloque.

En el cifrado en flujo cada bit del texto plano se cifra mediante el operador XOR con otro bit, obtenido de un generador pseudoaleatorio (típicamente un LFSR, como el de la **figura 4**). Algunos ejemplos de algoritmos de cifrado en bloque son A5, SEAL y RC4. Se puede ver un ejemplo en la **figura 5.**

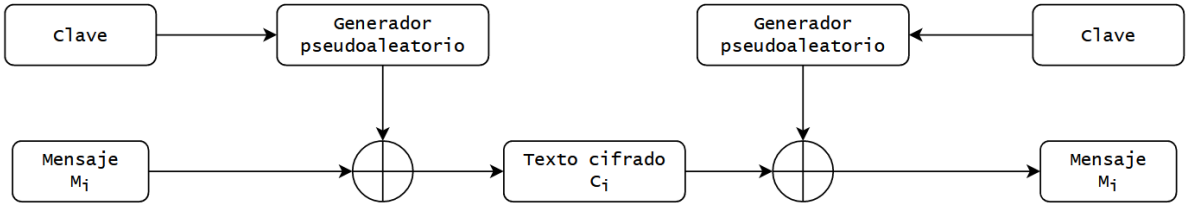
Figura 4. Esquema de ejemplo de LFSR de 4 celdas.



t	Registro	S_i
1	0001	1
2	1000	0
3	1100	0
4	1110	0
5	0111	1
6	0011	1
7	0001	

Si = 100011

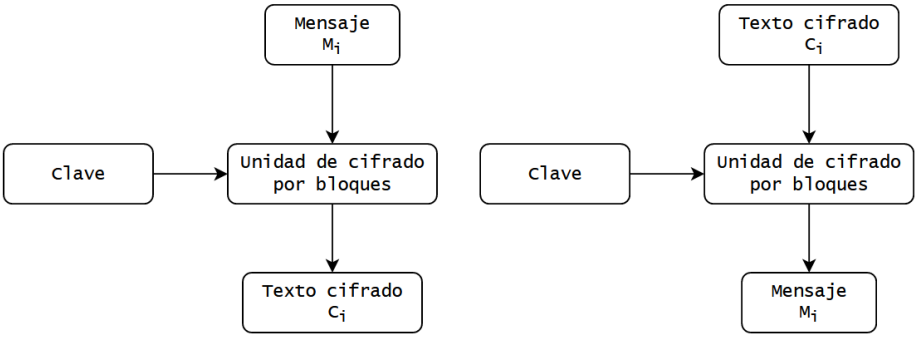
Figura 5. Esquema de cifrado en flujo.



Fuente: Elaboración propia.

En el cifrado en bloque, sin embargo, se cifra un bloque de tamaño fijo del texto original utilizando una clave. Esto se puede realizar con diferentes modos de operación (ECB, CBC, CTR, OFB, etc.), que hacen referencia a la influencia que tiene un bloque en los demás. Algunos ejemplos de algoritmos de cifrado en bloque son DES, AES y Blowfish. En la **figura 6** se puede ver un ejemplo de cifrado y descifrado.

Figura 6. Esquema de cifrado en bloque.



Fuente: Elaboración propia.

En la **tabla 2** se puede ver una comparativa entre algunos de los principales algoritmos. En ella se muestran algunas propiedades relevantes como los tamaños de clave y bloque, rondas que realiza, tipo de estructura del algoritmo y nivel de seguridad y eficiencia.

2.1.3. Funciones hash

Todos los algoritmos que se han comentado, ya sean de cifrado clásico o moderno, tienen un mecanismo que permite recuperar la información original que se ha cifrado. Sin embargo, existe un tipo de funciones en las cuales esto no es posible: las funciones hash.

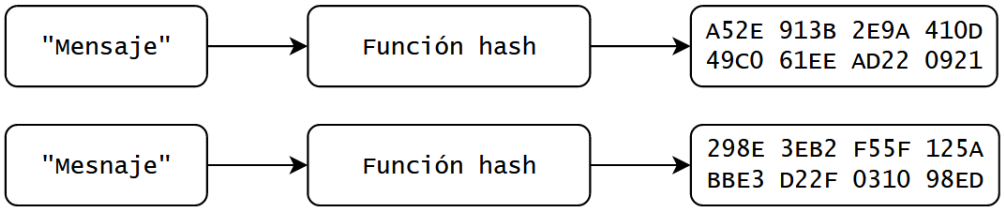
Este tipo de funciones convierte los datos que se le den como input en una cadena de texto de tamaño fijo. Una de las características más importantes de los hashes es el llamado efecto avalancha, lo que significa que cualquier modificación en el input, por pequeña que sea, cambie completamente el hash producido, como se puede ver en la **figura 7**.

Tabla 2. ***Comparativa de algoritmos criptográficos.***

	Año	Tamaño de clave (bits)	Tamaño de bloque (bits)	Rondas	Estructura	Flexible	Características
DES	1975	64	64	16	Feistel	No	Débil
DH	1976	Variable	-	-	Alg. de clave pública	Sí	Buena seguridad y lento
E-DES	1977	1024	128	16	Feistel	-	Buenas seguridad y rápido
RSA	1977	1024-4096	128	1	Alg. de clave pública	NO	Excelente seg. y lento
3DES	1978	112, 168	64	48	Feistel	Sí	Seg. adecuada y rápido
ECC	1985	Variable	Variable	1	Alg. de clave pública	Sí	Excelente seg. y rápido
EEE	1985	1024	-	-	Alg. de clave pública	Sí	Suficiente seg. y rápido
RC4	1987	Variable	40-2048	256	Flujo feistel	Sí	Cifrado rápido
RC2	1987	8, 64 (def), 128	64	16	Feistel	-	Buena y rápida seguridad
BLOWFISH	1993	32-448	64	16	Feistel	Sí	Cifrado rápido en SSL
SEAL	1994	160	32	2	Feistel	Sí	Débil y rápido
DSA	1997	Variable	-	-	Alg. de clave pública	Sí	Buena seguridad y rápido
RC6	1998	128 a 256	128	20	Feistel	Sí	Buena seguridad
AES	1999	128, 192, 156	128	10, 12, 14	Sustitución-permutación	Sí	El mejor en seguridad y rendimiento

Fuente: Obtenida de (Abood, 2018), adaptada y traducida

Figura 7. Efecto avalancha en una función hash



Fuente: Elaboración propia.

Esta no es la única propiedad que deben tener las funciones hash (Pittalia, 2019). Las dos principales son:

- **Resistencia débil a colisiones (o primera preimagen):** es computacionalmente imposible que, conocido un mensaje M, podamos encontrar un mensaje M' tal que h(M) = h(M').

- **Resistencia fuerte a colisiones (o segunda preimagen):** es computacionalmente imposible encontrar una pareja de mensajes M y M' de forma de que h(M) = h(M'). Si no se cumple, se podría realizar un ataque por la paradoja del cumpleaños.[2]

Las funciones hash se utilizan en muchos campos: firma digital (para proteger la integridad de un documento), sumas de verificación, derivación de claves (como en las OPT), detección de malware, guardar contraseñas en bases de datos (almacenando el hash en lugar de la contraseña), etc.

En la **tabla 3** se puede ver una comparativa del funcionamiento y rendimiento de las principales funciones de hash como son MD5, SHA-1, SHA-2, SHA-3 y Whirpool. Se pueden observar los tamaños de bloque, clave y hash, rondas, si se han encontrado colisiones, rendimiento, qué tipo de operaciones realizan, etc.

[2] Es una paradoja estadística que dice que, en un conjunto de 23 personas, hay un 50% de posibilidades de que dos de ellas cumplan años el mismo día. En criptografía esto afecta reduciendo la resistencia de preimagen de 2^n a $2^{n/2}$.

2.1.4. Criptografía en la actualidad

Durante los últimos años ha habido un gran incremento en la cantidad de ciberdelitos. En Estados Unidos en 2014, el 77% de las empresas reportaron algún tipo de incidente de seguridad y más el 34% dice que hay un incremento respecto al año anterior (PWC, 2014).

En España, como se puede ver en la **figura 8**, también ha habido un gran incremento en los ciberdelitos, principalmente de la categoría de fraude informático (el 89,6% de todos los ciberdelitos), en la cual se ha incrementado un 34,06% en 2020 respecto del año anterior (Ministerio del interior, 2021).

Tabla 3. Comparativa de funcionamiento y rendimiento de algoritmos de hash

Parámetros	**MD5**	**SHA-1**	**SHA-2**	**SHA-3**	**Whirpool**
Tamaño de bloque (bits)	512	512	512, 1024	1600-2*bits	512
Tamaño del resumen (bits)	128	160	160, 224, 256, 384, 512	160, 224, 256, 384, 512	512
Tamaño de palabra (bits)	32	32	32, 64	64	8
Rondas	4	80	80	24	10
Colisiones encontradas	Sí	Teórica	No	No	Sí
Operaciones	AND, OR, XOR, NOT	AND, OR, XOR, NOT	AND, OR, XOR, NOT, SHR	AND, OR, XOR, NOT, SHR	AND, OR, XOR, NOT
Rendimiento (ciclos/byte)[3]	4.77	1.89	SHA-224: 2.02 SHA-256: 2.02 SHA-384: 4.22 SHA-512: 4.22	SHA3-224: 5.07 SHA3-256: 5.31 SHA3-384: 6.93 SHA3-512: 9.98	13.84

Fuente: Obtenida de (Pittalia, 2019), adaptada y traducida

[3] amd64; Zen3 (a20f10); 2020 AMD Ryzen 9 5950X; 16 x 3400MHz; zen3, supercop-20220213

Figura 8. Evolución de ciberdelitos conocidos por categorías delictivas.

HECHOS CONOCIDOS	2016	2017	2018	2019	2020
ACCESO E INTERCEPTACIÓN ILÍCITA	3.243	3.150	3.384	4.004	4.653
AMENAZAS Y COACCIONES	12.036	11.812	12.800	12.782	14.066
CONTRA EL HONOR	1.546	1.561	1.448	1.422	1.550
CONTRA PROPIEDAD INDUST./INTELEC.	129	121	232	197	125
DELITOS SEXUALES(*)	1.231	1.392	1.581	1.774	1.783
FALSIFICACIÓN INFORMÁTICA	3.017	3.280	3.436	4.275	6.289
FRAUDE INFORMÁTICO	70.178	94.792	136.656	192.375	257.907
INTERFERENCIA DATOS Y EN SISTEMA	1.336	1.291	1.192	1.473	1.590
Total HECHOS CONOCIDOS	**92.716**	**117.399**	**160.729**	**218.302**	**287.963**

(*)Excluidos las agresiones sexuales con/sin penetración y los abusos sexuales con penetración

Fuente: (Ministerio del interior, 2021).

Estas crecientes amenazas son potenciales pérdidas económicas y reputación para las empresas. Aunque el principal factor es el error humano (Evans, 2016), (Yildirim, 2016), hay otros muchos elementos que influyen enormemente en la ciberseguridad, siendo uno de ellos el que se va a tratar en este trabajo: la criptografía.

2.1.5. Competición de funciones hash del NIST

En el año 2007 se anunció la competición de funciones hash del NIST con el objetivo de desarrollar una nueva función de hash llamada SHA-3, la cual sustituiría a SHA-1 y SHA-2 (este proceso se hizo de forma similar al realizado para desarrollar el algoritmo AES). (NIST, 2009)

Esta competición, en la cual participaron una gran cantidad de funciones, estuvo formada por tres rondas en la cuales se fueron descartando la mayoría en base a criterios de seguridad, rendimiento y características en la implementación.

A la final llegaron BLAKE, Grøstl, JH, Keccak y Skein (en la **tabla 4** se puede ver una comparativa de algunas de sus propiedades como sus tamaños de hash y primitiva, extensor de dominio y resistencia a colisiones), siendo seleccionado finalmente Keccak como ganador en 2012. Sin embargo, no fue declarado por el NIST como estándar hasta agosto de 2015.

BLAKE fue descartada debido a que su diseño se parecía demasiado al de SHA-2. Esto se debe a que lo que se buscaba era una función que sustituyera a SHA-2 en caso de que ésta fuera comprometida. Si la ganadora de SHA-3 se parecía en diseño a SHA-2, los criptoanálisis que se descubrieran para SHA-2 podrían ser usados también para SHA-3 (Chang, 2012).

Debido a diversos motivos comentados en (Aumasson, 2013), tras la competición los autores de BLAKE decidieron crear BLAKE2, una versión mejorada. Se comentará en detalle en los siguientes apartados.

Tabla 4. ***Características de seguridad de SHA2 y de los finalistas de SHA-3.***

Algoritmo	Extensor de dominio	Primitiva subyacente	Tamaño de la primitiva	Tamaño del hash	Seguridad			
					Colisión	Preimagen	2º preimagen	Indiff
BLAKE	HAIFA	Cifrado en bloque	k=512 b=512	224 256	112 128	224 256	224 256	128 128
			k=1024 b=1024	384 512	192 256	384 512	384 512	256 256
Grøstl	Grøstl	Un par de permutaciones	512 512	224 256	112 128	224 256	$256 - \log_2 L$	128 128
			1024 1024	384 512	192 256	384 512	512 – log2L	256 256
JH	JH	Permutación	1024	224 256 384 512	112 128 192 256	224 256 256 256	224 256 256 256	256 256 256 256
Kekkak	Esponja	Permutación	1600	224 256 384 512	112 128 192 256	224 256 384 512	224 256 384 512	224 256 384 512
Skein	UBI	Cifrado en bloque modificable	k=512 b=512 r=128	224 256 384 512	112 128 192 256	224 256 384 512	224 256 384 512	256 256 256 256
SHA-2	MD	Cifrado en bloque	k=512 b=256	224 256	112 128	224 256	$256 - \log_2 L$	1
			K=1024 b=512	384 512	192 256	384 512	$512 - \log_2 L$	1

Fuente: Obtenida y traducida de (Chang, 2012)

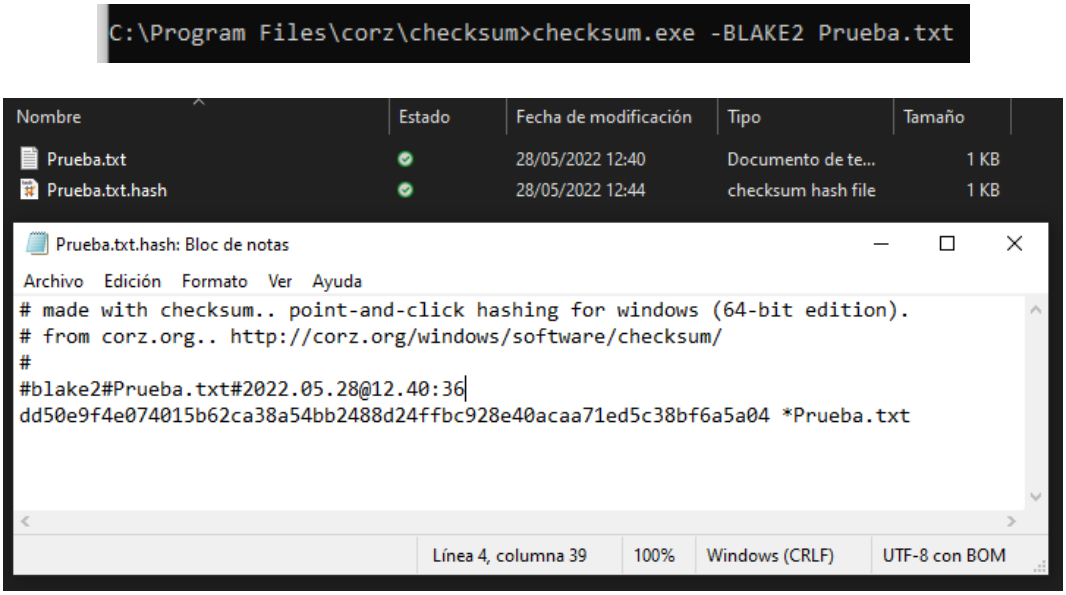
2.2. Herramientas ya existentes

Se ha realizado una revisión de las herramientas que ha sido posible encontrar para BLAKE2. Se han omitido bibliotecas que realizan el cálculo del hash (como hashlib). En la **tabla 5** se muestra una comparativa de estas herramientas.

2.2.1. checksum

Es una sencilla herramienta que funciona por línea de comandos y que permite generar y verificar hashes de cadenas de texto o archivos con las funciones MD5, SHA1 y BLAKE2. Solamente genera un archivo con el hash y algunos datos sobre la fecha en la que se hizo el hash, sin mostrar nada del proceso. En **la figura 9** se puede ver cómo se usa.

Figura 9. Uso del programa checksum



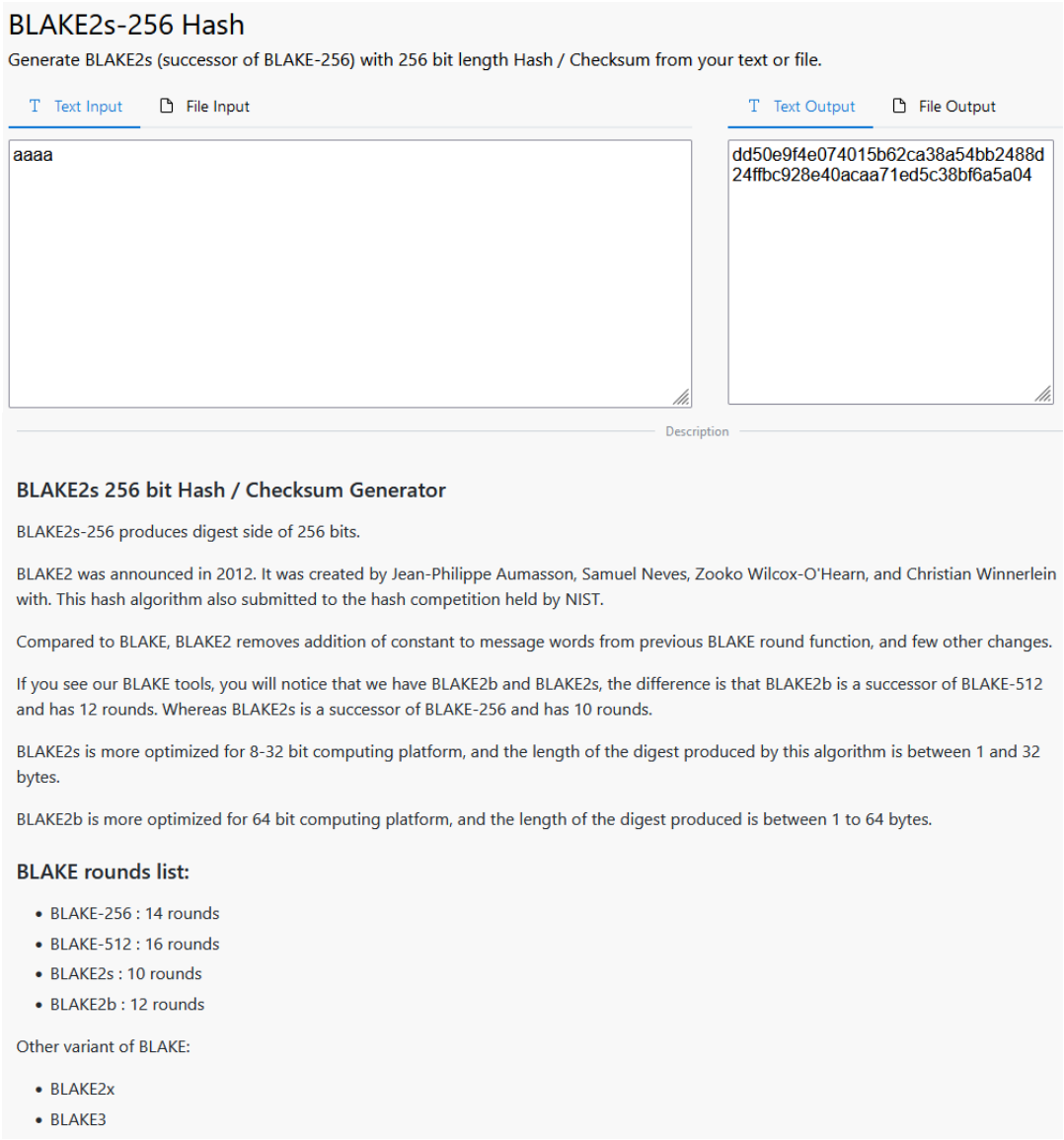
Fuente: Elaboración propia.

2.2.2. Toolkit Bay

Esta web permite obtener los hashes texto o archivos con una gran cantidad de funciones distintas: MD4, MD5, Adler 32, BLAKE2, BLAKE3, SHA-1, SHA-2 y SHA-3. Además, permite realizarlo con diferentes versiones de ellos (BLAKE2s-128, BLAKE2s-256, BLAKE2b-256, BLAKE2b-384, BLAKE2b-512, etc). No muestra nada de información del proceso, aunque muestra algo de información sobre el algoritmo.

En **la figura 10** se puede ver cómo se usa y qué información muestra de BLAKE2.

Figura 10. Uso del programa Toolkit Bay



Fuente: Elaboración propia.

2.2.3. blake2sum

Es una herramienta desarrollada en Python que, de forma similar a checksum, permite generar los hashes de texto o ficheros, en este caso con las funciones BLAKE2b y BLAKE2s. Sin embargo, no muestra nada del proceso de la función.

En la **figura 11** se puede ver el funcionamiento del programa.

Figura 11. Uso del programa blake2sum

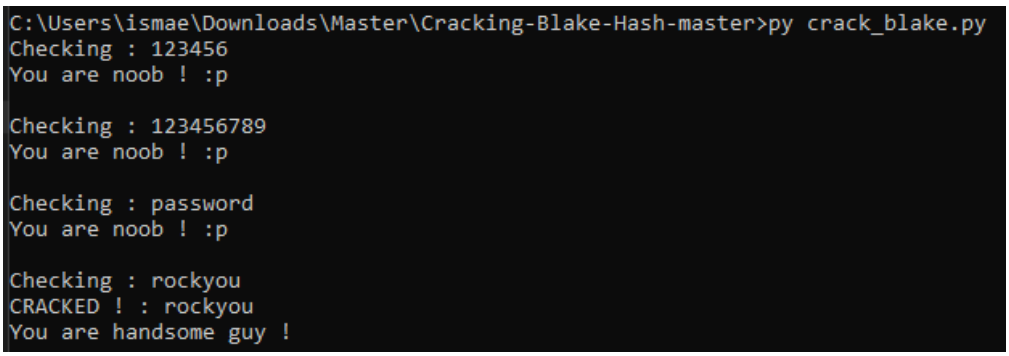
```
C:\Users\ismae\Downloads\Master\blake2sum-master\src>py blake2sum Prueba.txt
db467928862f7bdbe0897388193d7a17b3cd3b6f0cc63711ab1dbf31d1acc9297e20087c7b8bc0ca84639a2f
709070186bc0111597f953c58ccd8f208b44181a  Prueba.txt

C:\Users\ismae\Downloads\Master\blake2sum-master\src>py blake2sum -a blake2s Prueba.txt
dd50e9f4e074015b62ca38a54bb2488d24ffbc928e40acaa71ed5c38bf6a5a04  Prueba.txt
```

Fuente: Elaboración propia.

2.2.4. Cracking Blake

Esta herramienta permite obtener el hash de un texto (aunque hay que modificar el código para ello). Sin embargo, su principal función no es encriptar, sino intentar crackear un hash mediante fuerza bruta utilizando el famoso fichero rockyou.txt[4]. En la **figura 12** se puede ver un ejemplo de su uso para crackear un hash.

Figura 12. Uso del programa Cracking Blake



Fuente: Elaboración propia.

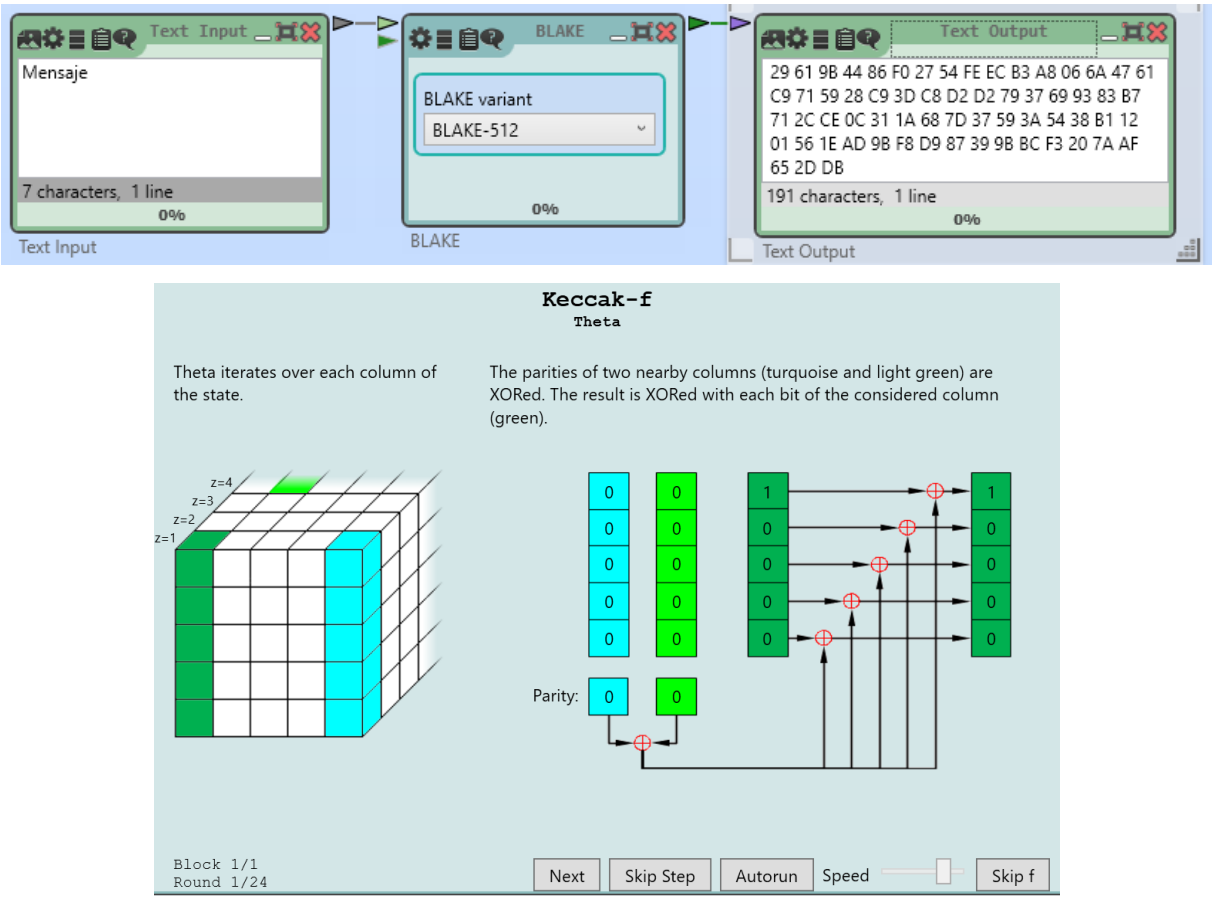
2.2.5. Cryptool 2

Cryptool 2 es una herramienta bastante completa y con interfaz de usuario que permite realizar una gran cantidad de operaciones distintas encadenando bloques para indicar el flujo del programa. Tiene operaciones de cifrado clásico (César, Enigma, etc.), moderno (AES, DES, RC4, Blowfish, etc), esteganografía, hashes (BLAKE, MD5, Keccak, Grøstl, etc.), criptoanálisis, etc.

Para algunos de ellos, como en el caso de Keccack, permite ver al detalle todos los pasos por los que pasa la función y cómo realiza todos los cálculos. Sin embargo, no permite realizar el cálculo de BLAKE2 (aunque sí de su predecesor, BLAKE, en muchas de sus variantes). En la **figura 13** se puede ver el funcionamiento del programa y cómo muestra la información del proceso interno del algortimo.

[4] *Rockyou.txt* es un archivo de texto con unas 14 millones de palabras utilizadas habitualmente como contraseñas. Se usa con herramientas como Hydra o John de Ripper para realizar ataques de fuerza bruta. Se puede descargar en GitHub.

Figura 13. Uso del programa Cryptool 2



Fuente: Elaboración propia.

2.3. Elección del algoritmo

En la **tabla 5** se puede observar una comparativa de las herramientas existentes, donde se muestran las propiedades más buscadas en estos, como son si permite visualizar los pasos de la función, lenguaje de implementación, si son de código abierto y si incluyen soporte teórico.

Aunque existe una gran cantidad de bibliotecas y algunas herramientas, estas no son muy detalladas respecto al proceso interno que realiza la función hash, de forma que no permiten poder estudiar de forma completa la función y sus fases.

Además, solo Cryptool 2 y Tookit Bay ofrecen algo de información sobre la función, aunque esta solamente hace referencia a algunos detalles de alto nivel de la función, sin entrar en detalle sobre el proceso interno.

Tabla 5. ***Comparativa de software de criptografía.***

	checksum	**Toolkit Bay**	**blake2sum**	**Cracking Blake**	**Cryptool 2**
Uso	Interfaz / línea de comandos	Web	Línea de comandos	Línea de comandos	Interfaz
Soporta BLAKE	No	No	No	Sí	Sí
Soporta BLAKE2	Sí	Sí	Sí	Sí	No
Crack	No	No	No	Sí	No
Visualización paso por paso	No	No	No	No	No
Incluye soporte teórico	No	Poco	No	No	Poco
Lenguaje de implementación	C	JavaScript	Python	Python	C#
Código fuente público	No	No	Sí	Sí	Sí

Fuente: Elaboración propia.

2.4. BLAKE2

2.4.1. Origen

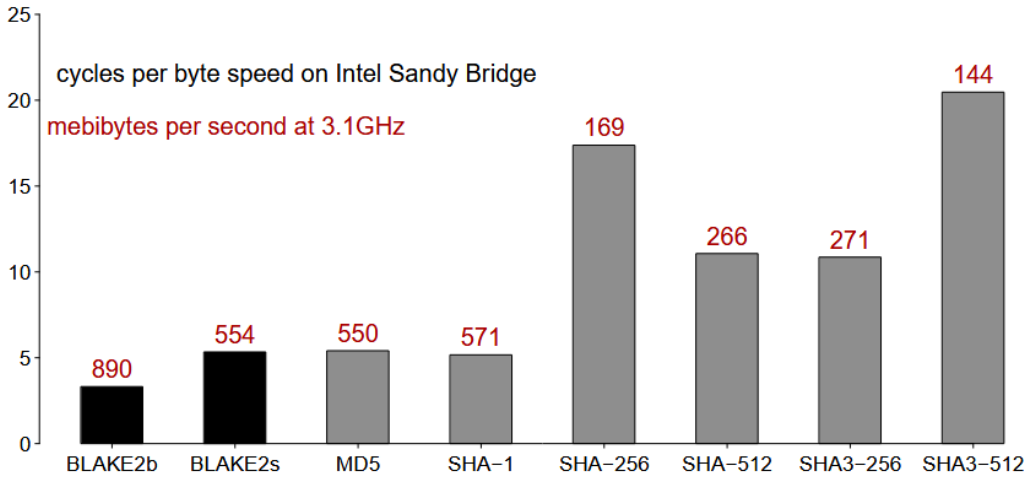
BLAKE2 es una función de hash basada en la propuesta para SHA-3, BLAKE. Fue anunciada a finales de 2012 y su objetivo era desarrollar una función que sustituyera a MD5 y SHA-1 (actualmente rotas), teniendo un rendimiento mejor que estas, pero con un mayor nivel de seguridad.

Para realizar esto, se hicieron una serie de cambios a BLAKE. Los principales son:

- Reducir el número de rondas (12 y 10 en lugar de 16 y 14).
- Constantes de rotación, relleno mínimo (más rápido y sencillo de implementar), menos constantes (8 en lugar de 24), little-endian.
- Modo paralelo, haciéndolo más rápido en CPUs multinúcleo o SIMD.
- Hashing en modo árbol para la actualización incremental o la verificación de archivos grandes.

Se entrará en detalle sobre ellos cuando se hable de la estructura de BLAKE2. En la **figura 14** se puede ver una comparativa de rendimiento de algunas de las principales funciones hash.

Figura 14. Comparativa de rendimiento de funciones hash.



Fuente: Obtenida de (Aumasson, 2013).

2.4.2. Estructura

Antes de entrar en detalle del funcionamiento de BLAKE2 es necesario indicar qué variables se utilizan durante el proceso y cuál es la notación que se va a seguir durante su explicación. Tanto estos elementos como el funcionamiento de la función en detalle han sido obtenidos de (Aumasson, 2013) para poder explicar su funcionamiento de la forma más clara y sencilla posible. En la **tabla 6** se muestran algunas características de BLAKE2b y BLAKE2s para poder ver sus diferencias (cantidad de rondas, tamaños de la entrada, clave, bloque, palabras y hash).

Tabla 6. ***Características y notación de BLAKE2b y BLAKE2s***

	BLAKE2b	**BLAKE2s**
Bits en una palabra	64 (8 bytes)	32 (4 bytes)
Rondas	12	10
Bytes por bloque	128	64
Bytes del hash	64	32
Bytes de la clave	0 - 64	0 - 32
Bytes de entrada	0 - 2^{128}	0 - 2^{64}

Fuente: Datos obtenidos de (Saarinen, 2015).

La notación que se seguirá en las funciones es la siguiente:

– **'←'** significa asignación a variable.

– **'+'** significa suma en $Z_{2^{32}}$ o en $Z_{2^{64}}$ (suma modular en base 2^{32} o 2^{64}).

– **'–'** significa resta en $Z_{2^{32}}$ o en $Z_{2^{64}}$ (resta modular en base 2^{32} o 2^{64}).

– **'⊕'** significa suma in Z_2^{32} or in Z_2^{64} (operador or exclusivo bit a bit).

– **'≪ r'** significa rotación de r bits hacia el bit más significativo.

– **'≫ r'** significa rotación de r bits hacia el bit menos significativo.

Si no se especifica lo contrario, los valores están en base 16 (F es 15).

Existen otras variables y constantes que se tienen que tener en cuenta para el cálculo del hash:

– **IV**: vector de inicialización (Son constantes. Después se explican su obtención).

– **σ**: sigma. Se usan para obtener permutaciones de palabras del mensaje (Son constantes. Ver **tabla 7**).

– **p**: bloque de parámetros (define parámetros del árbol de hashing y los tamaños del hash y la clave (si esta es usada).

– **m**: dieciséis palabras del mensaje (un bloque).

– **h**: estado del hash (o chaining value).

– **t**: offset de bytes del mensaje al final del bloque actual.

– **f**: flag que indica el último bloque (todo a unos si es el último).

Tabla 7. Permutaciones de σ.

	$\sigma_r[0]$	$\sigma_r[1]$	$\sigma_r[2]$	$\sigma_r[3]$	$\sigma_r[4]$	$\sigma_r[5]$	$\sigma_r[6]$	$\sigma_r[7]$	$\sigma_r[8]$	$\sigma_r[9]$	$\sigma_r[10]$	$\sigma_r[11]$	$\sigma_r[12]$	$\sigma_r[13]$	$\sigma_r[14]$	$\sigma_r[15]$
σ_0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ_1	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
σ_2	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
σ_3	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
σ_4	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
σ_5	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
σ_6	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
σ_7	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
σ_8	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
σ_9	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

Fuente: Datos obtenidos de (Saarinen, 2015).

Habiendo detallado esto, se procederá a explicar el desarrollo de la función. Para comenzar, uno de los elementos que se utilizan en la función es el **bloque de parámetros** (p). Su estructura se puede ver en la **figura 15 y 16.**

Figura 15. Estructura del bloque de parámetros de BLAKE2s.

Offset	0	1	2	3
0	Digest length	Key length	Fanout	Depth
4	Leaf length			
8	Node offset			
12	Node offset (cont.)		Node depth	Inner length
16 20	Salt			
24 28	Personalization			

Fuente: Obtenida de (Aumasson, 2013).

Figura 16. Estructura del bloque de parámetros de BLAKE2b.

Offset	0	1	2	3
0	Digest length	Key length	Fanout	Depth
4	Leaf length			
8 12	Node offset			
16	Node depth	Inner length	RFU	
20 24 28	RFU			
32 ... 44	Salt			
48 ... 60	Personalization			

Fuente: Obtenida de (Aumasson, 2013).

Se realizará un ejemplo para entenderlo mejor. Si se tienen los siguientes elementos:

- Output (Digest length) de 64 bytes. El valor del parámetro sería 40_{16}.
- Longitud de la clave (Key length) de 256 bits (32 bytes). El valor sería 20_{16}.
- La salt se establece con una cadena de 55_{16}.
- La personalización con una cadena de ee_{16}.
- *Fanout*, *depth*, *leaf length*, *node offset*, *node depth* e *inner length* son parámetros del árbol de hashing y no se entrará en detalle sobre ellos.

De esta forma, el bloque de parámetros tendría los valores que se pueden ver en la **figura 17** (cada espacio divide la cadena en bloques de 4 bytes para facilitar la lectura):

Figura 17. Ejemplo del cálculo del bloque de parámetros.

40200101 00000000 00000000 00000000 00000000 00000000 00000000 00000000

55555555 55555555 55555555 55555555 eeeeeeee eeeeeeee eeeeeeee eeeeeeee

Fuente: Obtenida de (Aumasson, 2013).

Por otro lado, tenemos los vectores de inicialización o **IVs**. Estos valores no son arbitrarios; hay una fórmula para obtenerlos. Sin embargo, en el cálculo del hash no se calculan estos valores, sino que se utilizan directamente. De todas formas, se explicará cómo se obtienen.

Cada elemento del vector de inicialización se obtiene con la siguiente fórmula:

$$IV[i]=\lfloor 2^{w} \cdot \{\sqrt{primo(i+1)}\} \rfloor$$

Donde el primo(i) es el i-ésimo número primo (2, 3, 5, 7, 11, 13, 17, 19, …) y la notación es:

- $\lfloor x \rfloor$ o floor(x): El mayor entero <= x. Algunos ejemplos:

  5,12 → 5

  8,98 → 8
- $\{x\}$ o frac(x): Parte fraccional positiva de x (frac(x) = x - floor(x)). Algunos ejemplos:

  5,12 → 0,12

  8,98 → 0,98

Por ejemplo, el cuarto IV para BLAKE2b sería el resultado de $\lfloor 2^{64} \cdot \{\sqrt{\text{primo}(7)}\} \rfloor$, que es 11.912.009.170.470.909.681 y tras pasarlo a hexadecimal obtenemos **A54FF53A5F1D36F1.**

De esta forma los IV para BLAKE2b serían:

0x6A09E667F3BCC908, 0xBB67AE8584CAA73B,

0x3C6EF372FE94F82B, **0xA54FF53A5F1D36F1**,

0x510E527FADE682D1, 0x9B05688C2B3E6C1F,

0x1F83D9ABFB41BD6B, 0x5BE0CD19137E2179

Y los IVs de BLAKE2s son:

0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A,

0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19

Los valores iniciales de **h** se obtienen combinando el bloque de parámetros con los IV utilizando la función xor: h ← p ⊕ IV.

Una vez hecho esto, se inicializan los valores de la función de compresión. $v_{0,\dots,}v_{15}$. Son lo que llamaremos **estado interno de la función**:

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ IV_0 & IV_1 & IV_2 & IV_3 \\ t_0 \oplus IV_4 & t_1 \oplus IV_4 & f_0 \oplus IV_6 & f_1 \oplus IV_7 \end{pmatrix}$$

Se inicializan con los siguientes elementos:

- **h_n**: ha sido calculado en el párrafo anterior.
- **t_0 y t_1**: es un contador de dos palabras que cuenta el número de bytes que ya han sido hasheados.
- **IV_n**: ya se ha explicado el proceso para obtenerlos.

- **f_0**: es todo ceros siempre que no sea el último bloque. Es decir, este valor será igual en todos los casos menos el último, en el cual se invierten todos los bits (pasa a ser todo unos).
- **f_1**: es igual que f_0, pero si se realiza el hashing en modo árbol, es utilizado también para indicar si es el último nodo de una capa. No se entrará en detalle sobre el modo árbol

Con los valores ya inicializados, a partir de este momento se empieza a realizar el cálculo del hash. Se comentará el proceso tanto para BLAKE2b como para BLAKE2s, comentando las pequeñas diferencias entre ellos.

Lo primero es dividir el mensaje original en bloques (de 128 bytes en BLAKE2b y de 64 bytes en BLAKE2s). En el caso del último bloque, si no es del tamaño de un bloque completo, se añaden ceros hasta completarlo.

Una vez hecho esto, se procede a realizar para cada bloque las rondas (12 en el caso de BLAKE2b y 10 en el de BLAKE2s), en las cuales se aplica la función G(a,b,c,d) ocho veces. Esta función tiene la siguiente estructura, variando ligeramente el desplazamiento dependiendo de si es BLAKE2b o BLAKE2s:

BLAKE2b
1 : a ← a + b + m_i
2 : d ← (d ⊕ a) ≫ *32*
3 : c ← c + d
4 : b ← (b ⊕ c) ≫ *24*
5 : a ← a + b + m_j
6 : d ← (d ⊕ a) ≫ *16*
7 : c ← c + d
8 : b ← (b ⊕ c) ≫ *63*

BLAKE2s
1 : a ← a + b + m_i
2 : d ← (d ⊕ a) ≫ *16*
3 : c ← c + d
4 : b ← (b ⊕ c) ≫ *12*
5 : a ← a + b + m_j
6 : d ← (d ⊕ a) ≫ *8*
7 : c ← c + d
8 : b ← (b ⊕ c) ≫ *7*

m_i y m_j son dos palabras del mensaje y se eligen de la siguiente forma:

- m_i es $\sigma_{r \ mod \ 10}(2i)$
- m_j es $\sigma_{r \ mod \ 10}(2i + 1)$

Donde i es el número de la G actual y r es el número de ronda actual, por lo que $\sigma_{r \ mod \ 10}$ es una de las 10 filas de la **tabla 7** e i indica las posiciones de esa fila. Estos valores empiezan a contar en cero, no en uno.

La función G ya comentada es aplicada 8 veces en cada ronda. 4 para las columnas:

$G_0(v_0, v_4, v_8, v_{12})$ $G_1(v_1, v_5, v_9, v_{13})$ $G_2(v_2, v_6, v_{10}, v_{14})$ $G_3(v_3, v_7, v_{11}, v_{15})$,

y 4 para las diagonales:

$G_4(v_0, v_5, v_{10}, v_{15})$ $G_5(\mathbf{v_1, v_6, v_{11}, v_{12}})$ $G_6(v_2, v_7, v_8, v_{13})$ $G_7(v_3, v_4, v_9, v_{14})$.

A continuación, se muestra un ejemplo con BLAKE2B. Supongamos que los valores iniciales son estos:

$$\begin{pmatrix} v_0 & \mathbf{v_1} & v_2 & v_3 \\ v_4 & v_5 & \mathbf{v_6} & v_7 \\ v_8 & v_9 & v_{10} & \mathbf{v_{11}} \\ \mathbf{v_{12}} & v_{13} & v_{14} & v_{15} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

Y suponemos que el mensaje es m[‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘H’, ‘I’, ‘J’, ‘K’, ‘L’, ‘M’, ‘N’, ‘O’, ‘P’] (en la práctica implicaría que después de cada elemento de *m* hay bytes nulos, hasta que el tamaño de cada elemento sea el de una palabra).

Como ya se comentó, m son 16 palabras (cada una de 64 bits u 8 bytes en BLAKE2b y de 32 bits u 4 bytes en BLAKE2s), que forman un bloque del mensaje (128 bytes en BLSKE2b y 64 bytes en BLAKE2s.

La primera llamada a G en la primera ronda sería G_0(a = 0, b = 4, c = 8, d = 12). Buscando los valores correspondientes en la **tabla 7** obtenemos m_i y m_j. Como es la primera función G, i = 0.

$m_i = \sigma_{0\ mod\ 10}(2 \cdot 0) = 0$. Obtenemos m_0, que es ‘A’ (valor 65)

$m_j = \sigma_{0\ mod\ 10}(2 \cdot 0 + 1) = 1$. Obtenemos m_1 que es ‘B’ (valor 66)

Ahora se realizan cada uno de los ocho pasos de la función G:

1: a ← a + b + m_i ⇒ a = 0 + 4 + 65 = 69.

a pasa de valer 0 a valer 69.

2: d ← (d ⊕ a) ≫ *32* ⇒ d es 12_{10}, en binario 0000 1100_2. a es 69_{10}, en binario 0100 0101_2 ⇒

La operación xor es 0000 1100_2 ⊕ 0100 0101_2 ⇒ 0100 1001_2.

Para el desplazamiento hay que tener en cuenta que se hace sobre el tamaño de la palabra, es decir, 64 bits. Esta operación quedaría de la siguiente forma:

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 **0100 1001** ⇒

0000 0000 0000 0000 0000 0000 **0100 1001** 0000 0000 0000 0000 0000 0000 0000 0000 =

= 313.532.612.608.

d pasa de valer 12 a valer 313.532.612.608.

3: c ← c + d ⇒ c = 8 + 313.532.612.608 = 313.532.612.616

c pasa de valer 8 a valer 313.532.612.616.

4: b ← (b ⊕ c) ≫ *24* ⇒

b es 4_{10}, en binario 0000 0100_2.

c es 313.532.612.616_{10}, en binario 0100 1001 0000 0000 0000 0000 0000 0000 0000 1000_2.

La operación xor es 0100 1001 0000 0000 0000 0000 0000 0000 0000 1100_2.

A continuación, hacemos el desplazamiento igual que en el paso 2, esta vez en 24:

0000 0000 0000 0000 0000 0000 **0100 1001** 0000 0000 0000 0000 0000 0000 0000 **1100** ⇒

0000 0000 0000 0000 0000 **1100** 0000 0000 0000 0000 0000 0000 **0100 1001** 0000 0000 =

= 13.194.139.552.000

b pasa de valer 4 a valer 13.194.139.552.000

5: a ← a + b + mj ⇒ a = 69 + 13.194.139.552.000 + 66 = 13.194.139.552.135

a pasar de valer 69 a valer 13.194.139.552.135.

6: d ← (d ⊕ a) ≫ 16 ⇒

d es 313.532.612.608, en bin 0100 1001 0000 0000 0000 0000 0000 0000 0000 0000_2

a es 13.194.139.552.135, en bin 1100 0000 0000 0000 0000 0000 0000 0100 1001 1000 0111_2

La operación xor es 1100 0100 1001 0000 0000 0000 0000 0100 1001 1000 0111_2

A continuación, hacemos el desplazamiento igual que en los pasos anteriores, esta vez en 16:

0000 0000 0000 0000 0000 **1100 0100 1001** 0000 0000 0000 0000 **0100 1001 1000 0111** ⇒

0100 1001 1000 0111 0000 0000 0000 0000 0000 **1100 0100 1001** 0000 0000 0000 0000 =

= 5.298.203.486.830.788.608

d pasa de valer 313.532.612.608 a valer 5.298.203.486.830.788.608

7: c ← c + d ⇒ c = 313.532.612.616 + 5.298.203.486.830.788.608 = 5.298.203.800.363.401.224

c pasa de valer 313.532.612.616 a valer 5.298.203.800.363.401.224.

8: b ← (b ⊕ c) ≫ 63 ⇒

b es 1100 0000 0000 0000 0000 0000 0000 0100 1001 0000 0000

c es 0100 1001 1000 0111 0000 0000 0100 1001 0000 1100 0100 1001 0000 0000 0000 1000

xor: 0100 1001 1000 0111 0000 1100 0100 1001 0000 1100 0100 1001 0100 1001 0000 1000

A continuación, hacemos el desplazamiento igual que en los pasos anteriores, esta vez en 63:

0100 1001 1000 0111 0000 1100 **0100 1001** 0000 1100 0100 1001 0100 1001 0000 1000 ⇒

1001 0011 0000 1110 0001 100**0 1001 001**0 0001 1000 1001 0010 1001 0010 0001 0000 =

= 10.596.433.989.005.906.448

b pasa de valer 13.194.139.552.000 a valer 10.596.433.989.005.906.448.

De esta forma ya tendríamos el primer resultado:

a = 13.194.139.552.135 c = 5.298.203.800.363.401.224

b = 10.596.433.989.005.906.448 d= 5.298.203.486.830.788.608

Ahora habría que repetir G para el resto de columnas y las diagonales. Tras haber hecho eso, se obtendría el resultado de la primera ronda, por lo que habría que repetir todo el proceso hasta realizar las 12 rondas (10 en BLAKE2s).

Por último, se realizan unas últimas operaciones combinando los valores del *estado interno* con el estado actual del hash:

$h'_0 \leftarrow h_0 \oplus v_0 \oplus v_8$ $h'_4 \leftarrow h_4 \oplus v_4 \oplus v_{12}$

$h'_1 \leftarrow h_1 \oplus v_1 \oplus v_9$ $h'_5 \leftarrow h_5 \oplus v_5 \oplus v_{13}$

$h'_2 \leftarrow h_2 \oplus v_2 \oplus v_{10}$ $h'_6 \leftarrow h_6 \oplus v_6 \oplus v_{14}$

$h'_3 \leftarrow h_3 \oplus v_3 \oplus v_{11}$ $h'_7 \leftarrow h_7 \oplus v_7 \oplus v_{15}$

O lo que sería lo mismo:

Desde n = 0 hasta 8 ⇒ $h'_n \leftarrow h_n \oplus v_n \oplus v_{n+8}$

Todo lo realizado anteriormente se repite para cada bloque. El output final de la función de compresión del último bloque es el hash (64 o 32 bytes dependiendo de la versión).

2.5. Conclusiones del estado del arte

Durante el estudio del estado del arte se ha podido observar la carencia de herramientas educativas para la función BLAKE2 ya que, si bien existen una gran cantidad de bibliotecas y bastantes herramientas que lo utilizan, ninguna de ellas permite entender cómo se realiza este proceso.

3. Objetivos

Tras haber presentado el algoritmo BLAKE2, sus propiedades y su funcionamiento, así como las herramientas que existen para su estudio y aprendizaje, en este capítulo se procederá a determinar los objetivos que persigue el desarrollo del software educativo a través de este TFM.

Se comenzará determinando el objetivo general y, a continuación, objetivos específicos que, en su conjunto, permitirán lograr el objetivo general. Por último, se explicará la metodología de trabajo, el marco tecnológico y la planificación del desarrollo.

3.1. Objetivo general

Desarrollo de un software con interfaz gráfica que facilite a los estudiantes el aprendizaje del funcionamiento del algoritmo BLAKE2. Debe permitir poder realizar un seguimiento de forma sencilla de los procesos internos que se van realizando durante los diferentes pasos del algoritmo, permitiendo retroceder y avanzar en los mismos para facilitar su comprensión y seguimiento.

3.2. Objetivos específicos

- **Estudio el algoritmo BLAKE2**: es necesario comprender al detalle cómo funciona el algoritmo y todas sus fases para poder tener una gran comprensión del funcionamiento del mismo y posteriormente poder desgranarlo para facilitar su comprensión.

- **Análisis y comparativa de las herramientas criptográficas existentes**: de esta forma se pretende poder comprender las capacidades y carencias de las herramientas existente con el objetivo de determinar qué mejoras o modificaciones se pueden realizar de forma que se aumente el valor del software final.

- **Análisis y valoración de las posibles alternativas para desarrollar el software**: de forma que la solución elegida sea factible en términos de tiempo y recursos de desarrollo y a su vez permita obtener una herramienta lo más completa posible y cumpla lo máximo posible con el objetivo general.

- **Elección de una metodología de trabajo:** una metodología que se adapte con precisión a las características y requisitos del proyecto.

- **Diseño de una interfaz de usuario**: que sea fácil de entender por estudiantes, pero a su vez permita ver el proceso del hash en detalle.

3.3. Método de trabajo

Como todo proyecto de ingeniería, es necesario el uso de una metodología que permita abordar de manera óptima los objetivos propuestos para el mismo. Por ello, en este capítulo se detallará la metodología que ha sido utilizada para el desarrollo y cómo se ha aplicado.

3.3.1. Metodología de desarrollo

Para la elección de la metodología se han tenido en cuenta la propia naturaleza del proyecto, sus requisitos y sus objetivos.

Los elementos que más se han tenido en cuenta durante la elección han sido los siguientes:

- Posibilidad de añadir nuevas funcionalidades a la vez que se realizan correcciones en componentes anteriormente implementadas.
- Posibilidad del surgimiento de nuevos requisitos durante el proceso de implementación del software.
- El desarrollo es realizado por una sola persona.
- Proyecto de corta duración.

Además, con el objetivo de minimizar el impacto de los cambios durante el desarrollo, se han realizado reuniones de forma periódica para verificar el correcto desarrollo del proyecto, detectar posibles problemas o mejoras posibles y determinar los siguientes pasos del desarrollo.

Teniendo en cuenta todos estos factores, se ha decidido seguir una metodología ágil basada en el proceso de *desarrollo iterativo e incremental*. A continuación, para sustentar esta decisión y poner en contexto, se comentará brevemente cuáles son sus características y las diferencias entre las metodologías tradicionales y las ágiles. Las principales diferencias pueden verse en la **tabla 8**, donde se comparan propiedades como facilidad de adaptación, duración, tamaño de los equipos, etc. Se ha utilizado como base para la elección de la metodología.

Tabla 8. ***Diferencias entre metodologías tradicionales y ágiles.***

	Tradicional	**Ágil**
TAMAÑO	Proyectos de cualquier tamaño	Proyectos más pequeños
ADAPTACIÓN	Problemas de adaptabilidad en proyectos pequeños	Problemas de adaptabilidad en proyectos grandes
TAMAÑO EQUIPO	Equipos grandes	Equipos pequeños
DURACIÓN	Cualquier duración	Corta duración
DOCUMENTACIÓN	Mucha documentación	Poca documentación
ROLES	Roles específicos	Roles más genéricos
CONTRATO	Prefijado	Flexible
ARQUITECTURA	Prefijada	Se redefine y mejora
CAMBIOS	No se esperan grandes cambios	Se esperan cambios
CONTROL	Control de cambios estrictos	Control menor

Fuente: Elaboración propia.

En 2001, CEOs de grandes empresas de Utah se reunieron y crearon *The Agile Alliance* (una organización cuyo objetivo era el de promover los valores de las metodologías ágiles) y el manifiesto ágil. Este resume la metodología ágil:

- Individuos e interacciones sobre procesos y herramientas.
- Software funcionando sobre documentación extensiva.
- Colaboración con el cliente sobre negociación contractual.
- Respuesta ante el cambio sobre seguir un plan.

Indicando que, aunque son importantes los elementos de la derecha, se valoran más los de la izquierda (Fowler, 2001). Algunas ventajas que ofrecen las metodologías ágiles sobre las tradicionales son:

- Desarrollar software más rápido y poder mostrar al cliente los avances del proyecto de manera más sencilla.
- Reducir los errores y encontrarlos más rápidamente.
- Adaptar el proyecto rápidamente a posibles cambios durante el desarrollo.
- Menor cantidad de documentación.

Sin embargo, tiene también sus inconvenientes, por ejemplo:

- Es difícil determinar con precisión el tiempo y dinero que requerirá el desarrollo.
- La escasa documentación puede complicar la entrada de nuevos desarrolladores.
- La falta de definición del alcance del proyecto puede hacer que este crezca de forma descontrolada (scope creep).
- Requiere de un alto nivel de interacción entre los desarrolladores y los clientes, cosa que no siempre es posible.

Además, es importante explicar los puntos clave de esta metodología (Larman, 2003), junto con su aplicación en el desarrollo del proyecto.

Desarrollo incremental

Es una estrategia de desarrollo que permite que distintas partes del sistema pueden ser desarrolladas paralelamente o en diferentes momentos mediante la división del proyecto en distintos componentes, los cuales se pueden entregar al cliente de forma independiente y al final del desarrollo ser integrados en el sistema final, haciendo que el producto evolucione a lo largo de las iteraciones.

Desarrollo iterativo

Hace referencia a que las distintas partes del sistema son revisadas para poder hacer mejoras o corregir los errores que hayan podido surgido en iteraciones anteriores. Además, cada iteración produce una versión funcional del producto, desarrollándolo poco a poco y permitiendo obtener feedback continuo por parte del cliente y permitiendo reducir los riesgos, ya que se pueden detectan más fácilmente los problemas en las fases tempranas del desarrollo.

El software desarrollado en este TFM se ha realizado siguiendo esta metodología para poder dividir, revisar y modificar los componentes desarrollados, así como detectar, corregir errores y permitir la realización de un sistema escalable y robusto.

3.3.2. Marco tecnológico

En este apartado se describirán los medios software, que han sido utilizados durante el proceso de desarrollo.

Herramientas de desarrollo:

- **Visual studio code**: un IDE de libre uso desarrollado por Microsoft. Incluye soporte para la depuración, control de versiones con Git, resaltado de sintaxis, uso de snippets4, refactorización, etc. Este entorno es muy configurable y permite incluir extensiones para añadir funcionalidades adicionales.

- **Trello**: es un software de administración de proyectos de manera colaborativa con interfaz web y cliente para iOS y Android. Permite crear tableros con listas y tarjetas virtuales con el objetivo de organizar los requisitos y los hitos de un proyecto.

- **Github**: es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. En él se ha alojado el código del proyecto.

Documentación:

- **Microsoft word**: es un programa de edición de texto creado por Microsoft. Con él se ha creado toda la documentación del proyecto y se ha dado soporte a la revisión del contenido por parte de lo tutor del TFM.

- **Draw.io**: es una herramienta online de Google e integrada con Google Drive que permite crear bocetos y diagramas de forma colaborativa. Se ha usado para realizar muchas de las figuras de la memoria.

- **Balsamiq Mockups**: es una herramienta que permite generar bocetos de interfaz de usuario (IU) utilizando widgets ya creados mediante un editor WYSIWYG[5]. Se ha utilizado para realizar los diseños de las interfaces de usuario.

[5] WYSIWYG, acrónimo de What You See Is What You Get (en español, "lo que ves es lo que obtienes"), hace referencia a editores que permiten escribir un documento mostrando directamente el resultado final.

Bibliotecas:

- **Haslib**: permite realizar el cálculo de hashes con una gran cantidad de algoritmos diferentes. Se ha utilizado para realizar los test y comprobar el correcto funcionamiento del software desarrollado.

- **Tkinter (tcl/tk):** permite desarrollar aplicaciones de escritorio multiplataforma (Windows, Linux, Mac, etc.). Está escrita en C, es de código abierto y fue desarrollada en sus orígenes para el lenguaje de programación Tcl. Viene instalado con Python, es fácil de aprender y cuenta con una documentación muy completa. Se ha utilizado para el desarrollo de la interfaz completa del software.

Lenguajes de programación:

- **Python 3.8**: es un lenguaje de programación interpretado, muy flexible, ampliamente utilizado y con una gran cantidad de bibliotecas con funcionalidades matemáticas y de interfaces gráficas, etc. El software se ha desarrollado completamente con Python.

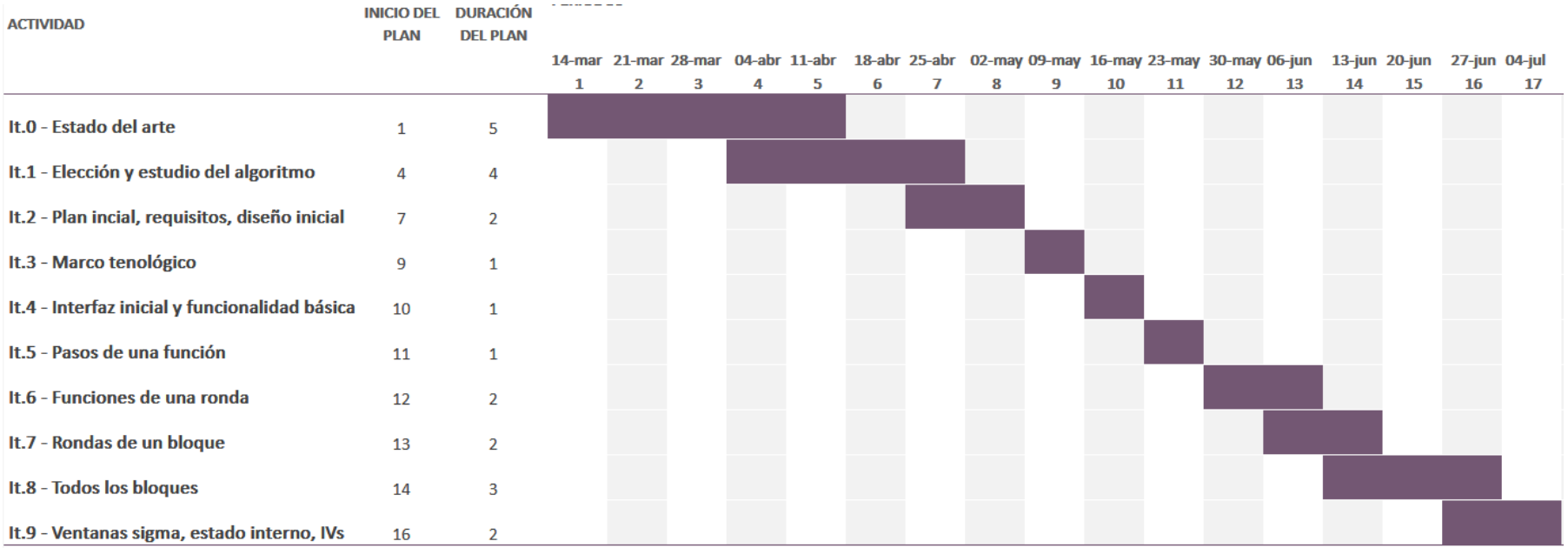
3.3.3. Planificación

Como ya se ha comentado, el desarrollo del proyecto está dividido en iteraciones, las cuales a su vez están divididas en subtareas (se detallarán en el capítulo 5):

- **Iteración 0**: Estudio del estado del arte, algoritmos y herramientas existentes.
- **Iteración 1**: Elección del algoritmo y estudio del mismo.
- **Iteración 2**: Elaboración del plan inicial con el alcance del proyecto, requisitos, planificación y desarrollo de bocetos y diagramas.
- **Iteración 3**: Elección del marco tecnológico, instalación y configuración de las herramientas a utilizar durante el desarrollo.
- **Iteración 4**: Implementación de la interfaz inicial y funcionalidades básicas del software.
- **Iteración 5**: Visualización de todos los pasos de una función G.
- **Iteración 6**: Visualización de todas las funciones de una ronda.
- **Iteración 7**: Visualización de todas las rondas de un bloque.
- **Iteración 8**: Visualización de todos los bloques de la función hash.
- **Iteración 9**: Ventanas de sigma, estado interno e IVs.

En la **figura 18** se muestra el diagrama de Gantt correspondiente a la distribución de tareas realizadas a lo largo del desarrollo.

Figura 18. Diagrama de Gantt



Fuente: Elaboración propia.

4. Análisis y diseño de la aplicación

En este apartado se especificarán los requisitos funcionales y no funcionales, junto a una breve descripción de los mismos. De esta forma, el objetivo es identificar las necesidades que el software debe satisfacer para cumplir con los objetivos definidos en el capítulo 3. Además, se incluirán los bocetos y diagramas iniciales.

4.1. Requisitos

4.1.1. Requisitos funcionales

- El usuario debe poder ingresar cualquier cadena de texto, del cual se realizará el hash.
- El sistema debe mostrar los pasos de cada función G.
- El sistema debe mostrar todas las funciones de cada ronda.
- El sistema debe mostrar todas las rondas de cada bloque.
- El sistema debe mostrar todos los bloques procesados por la función.
- El sistema debe mostrar toda la información adicional que participe en el proceso del hash y que permita seguir este proceso.

4.1.2. Requisitos no funcionales

- La presentación de los pasos de la función se debe realizar de la forma más sencilla y clara posible, de forma que sea sencillo seguir el proceso interno de la función.
- El sistema debe ser escalable y robusto.
- El sistema debe ser una aplicación de escritorio portable con GUI.
- El sistema debe ofrecer una guía de uso.

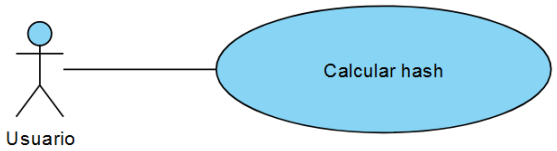
4.2. Casos de uso

Una vez definidos los requisitos, el siguiente paso es diseñar los casos de uso. Un caso de uso es una descripción de una secuencia de interacciones que se realiza entre el sistema y el usuario final, como respuesta a un evento inicial del usuario con el sistema. Por último, se dispondrá de un diagrama de casos de uso para el comportamiento de forma global del sistema con los usuarios.

4.2.1. Calcular hash

El usuario introduce una cadena de texto que será sobre la que se aplicará la función hash y sobre la que se mostrarán los al detalle todos los pasos internos que realiza la función.

Figura 19. Caso de uso Calcular hash.



Fuente: Elaboración propia.

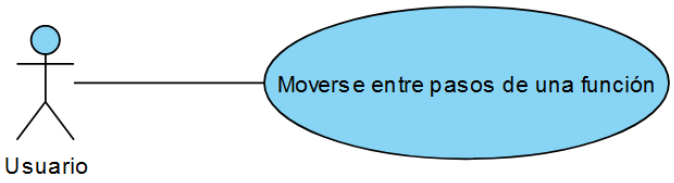
Tabla 9. *Detalles del caso de uso de Calcular hash.*

Nº	**CU1**
Nombre	Calcular hash
Descripción	Realización del cálculo del hash sobre la cadena de texto introducida por el usuario
Actores	Usuario
Precondiciones	Ninguna
Flujo de eventos	1. El usuario elige “BLAKE2” dentro del menú de “operaciones”. 2. El usuario introduce la cadena de texto a procesar. 3. El sistema valida los datos introducidos. 4. El sistema realiza el cálculo del hash. 5. El sistema carga las ventanas con la información sobre el inicio del proceso interno de la función. 6. El usuario puede: 6.1. Moverse a lo largo de los diferentes cálculos de la función. 6.2. Resetear los cálculos para introducir una cadena nueva.
Flujo optativo	O1: el usuario no ha introducido una cadena de texto: O1.1. El sistema indica al usuario que debe introducir una cadena.
Postcondiciones	Se realiza el cálculo del hash. Se guardan internamente los cálculos realizados. Se abren las distintas ventanas necesarias para la representación de los pasos internos de la función.

4.2.2. Moverse entre pasos de una función

El usuario puede desplazarse entre los pasos de una función para poder ver en detalle cuáles son los cálculos que se han realizado en cada uno de los pasos.

Figura 20. Caso de uso Moverse entre pasos de una función.



Fuente: Elaboración propia.

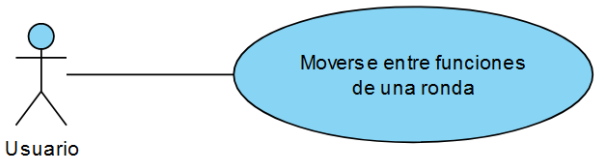
Tabla 10. ***Detalles del caso de uso de Moverse entre pasos de una función.***

Nº	**CU2**
Nombre	Moverse entre pasos de una función
Descripción	Acción para desplazarse al paso anterior o siguiente dentro de la función G que está cargada en ese momento.
Actores	Usuario
Precondiciones	CU1
Flujo de eventos	1. El usuario se mueve: 1.1. al paso anterior de la función. 1.2. al paso siguiente de la función. 2. El sistema carga la información de ese paso.
Flujo optativo	O1: no posible desplazarse ya que el usuario: O1.1. ya está en el primer paso. O1.2. ya está en el último paso. O2: se resetean los cálculos para introducir una nueva cadena.
Postcondiciones	Se carga la información del paso correspondiente (cálculo de a, b, c o d, dependiendo del paso que se esté cargando) y se actualiza la información del resto de ventanas si es necesario.

4.2.3. Moverse entre funciones de una ronda

El usuario puede desplazarse entre las funciones de una ronda para poder ver en detalle cuáles son los cálculos que se han realizado en cada una de las funciones.

Figura 21. Caso de uso Moverse entre funciones de una ronda.



Fuente: Elaboración propia.

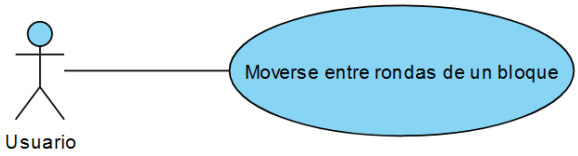
Tabla 11. ***Detalles del caso de uso de Moverse entre funciones de una ronda.***

Nº	**CU3**
Nombre	Moverse entre pasos de una función
Descripción	Acción para desplazarse al paso anterior o siguiente dentro de la función G que está cargada en ese momento.
Actores	Usuario
Precondiciones	CU1
Flujo de eventos	1. El usuario se mueve: 1.1. a la función anterior de la ronda. 1.2. a la función siguiente de la ronda. 2. El sistema carga la información de esa función. 3. El sistema carga la información del primer paso de esa función.
Flujo optativo	O1: no posible desplazarse ya que el usuario: O1.1. ya está en la primera función. O1.2. ya está en la última función. O2: se resetean los cálculos para introducir una nueva cadena.
Postcondiciones	Se carga la información de la función correspondiente (a, b, c y d finales de esa función) y de cada uno de sus pasos. Se actualiza la información del resto de ventanas si es necesario.

4.2.4. Moverse entre rondas de un bloque

El usuario puede desplazarse entre las rondas de un bloque para poder ver en detalle cuáles son los cálculos que se han realizado en cada una de las rondas.

Figura 22. Caso de uso Moverse entre rondas de un bloque.



Fuente: Elaboración propia.

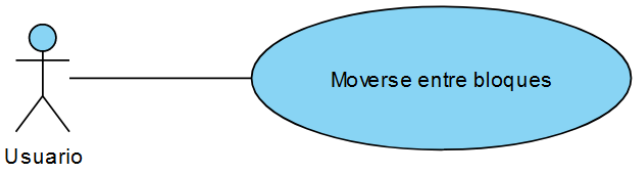
Tabla 12. ***Detalles del caso de uso de Moverse entre rondas de un bloque.***

Nº	**CU4**
Nombre	Moverse entre rondas de un bloque.
Descripción	Acción para desplazarse a la ronda anterior o siguiente dentro del bloque que está cargado en ese momento.
Actores	Usuario
Precondiciones	CU1
Flujo de eventos	1. El usuario se mueve: 1.1. a la ronda anterior de un bloque. 1.2. a la ronda siguiente de un bloque. 1. El sistema carga la información de esa ronda. 2. El sistema carga la información de la primera función de esa ronda. 3. El sistema carga la información del primer paso de la primera función.
Flujo optativo	O1: no posible desplazarse ya que el usuario: O1.1. ya está en la primera ronda. O1.2. ya está en la última ronda. O2: se resetean los cálculos para introducir una nueva cadena.
Postcondiciones	Se carga la información de la ronda correspondiente (todos los a, b, c y d finales de esa ronda) y de cada una de sus funciones G. Se actualiza la información del resto de ventanas si es necesario.

4.2.5. Moverse entre bloques

El usuario puede desplazarse entre los bloques (si hay más de uno) para poder ver en detalle cuáles son los cálculos que se han realizado en cada uno de los bloques.

Figura 23. Caso de uso Moverse entre bloques.



Fuente: Elaboración propia.

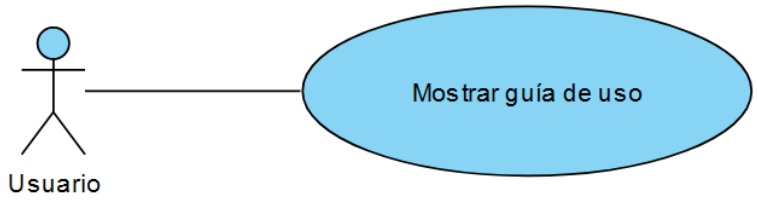
Tabla 13. ***Detalles del caso de uso de Moverse entre bloques.***

Nº	**CU5**
Nombre	Moverse entre bloques.
Descripción	Acción para desplazarse al bloque anterior o siguiente de la función.
Actores	Usuario
Precondiciones	CU1
Flujo de eventos	1. El usuario se mueve: 1.1. al bloque anterior. 1.2. al bloque siguiente. 2. El sistema carga la información de ese bloque. 3. El sistema carga la información de la primera ronda de ese bloque. 4. El sistema carga la información de la primera función de esa ronda. 5. El sistema carga la información del primer paso de la primera función.
Flujo optativo	O1: no posible desplazarse ya que el usuario: O1.1. ya está en el primer bloque. O1.2. ya está en el último bloque. O2: se resetean los cálculos para introducir una nueva cadena.
Postcondiciones	Se carga la información del bloque correspondiente (h finales de ese bloque) y de cada una de sus rondas. Se actualiza la información del resto de ventanas si es necesario.

4.2.6. Mostrar guía de uso

Debido a que realizar el seguimiento de los procesos internos de la función es difícil y contiene mucha información, se ha añadido una función de guía de uso que permita facilitar el seguimiento de la información mostrada.

Figura 24. Caso de uso Mostrar guía de uso.



Fuente: Elaboración propia.

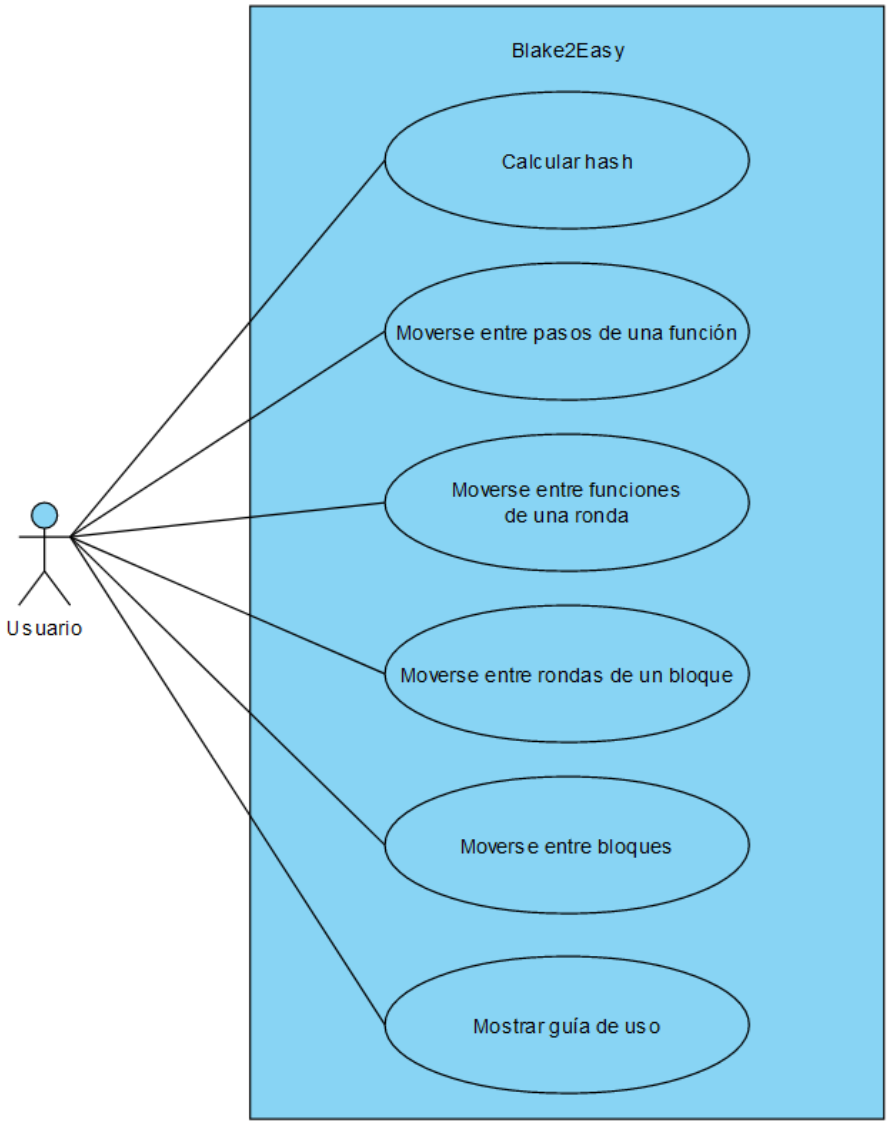
Tabla 14. ***Detalles del caso de uso de Mostrar guía de uso.***

Nº	**CU6**
Nombre	Mostrar guía de uso.
Descripción	El sistema muestra información sobre el funcionamiento del programa para facilitar el seguimiento del mismo por parte del usuario.
Actores	Usuario
Precondiciones	Ninguna
Flujo de eventos	1. El usuario elige “Guía de uso” dentro del menú de “Ayuda”. 2. El sistema crea una ventana nueva y carga la información de la guía de uso.
Flujo optativo	No hay
Postcondiciones	Se crea una ventana con la información.

Diagrama de casos de uso

En la **figura 25** se muestra el diagrama de casos de uso, teniendo en cuenta que el único rol soportado en el sistema es el de usuario final genérico.

Figura 25. Diagrama de casos de uso.

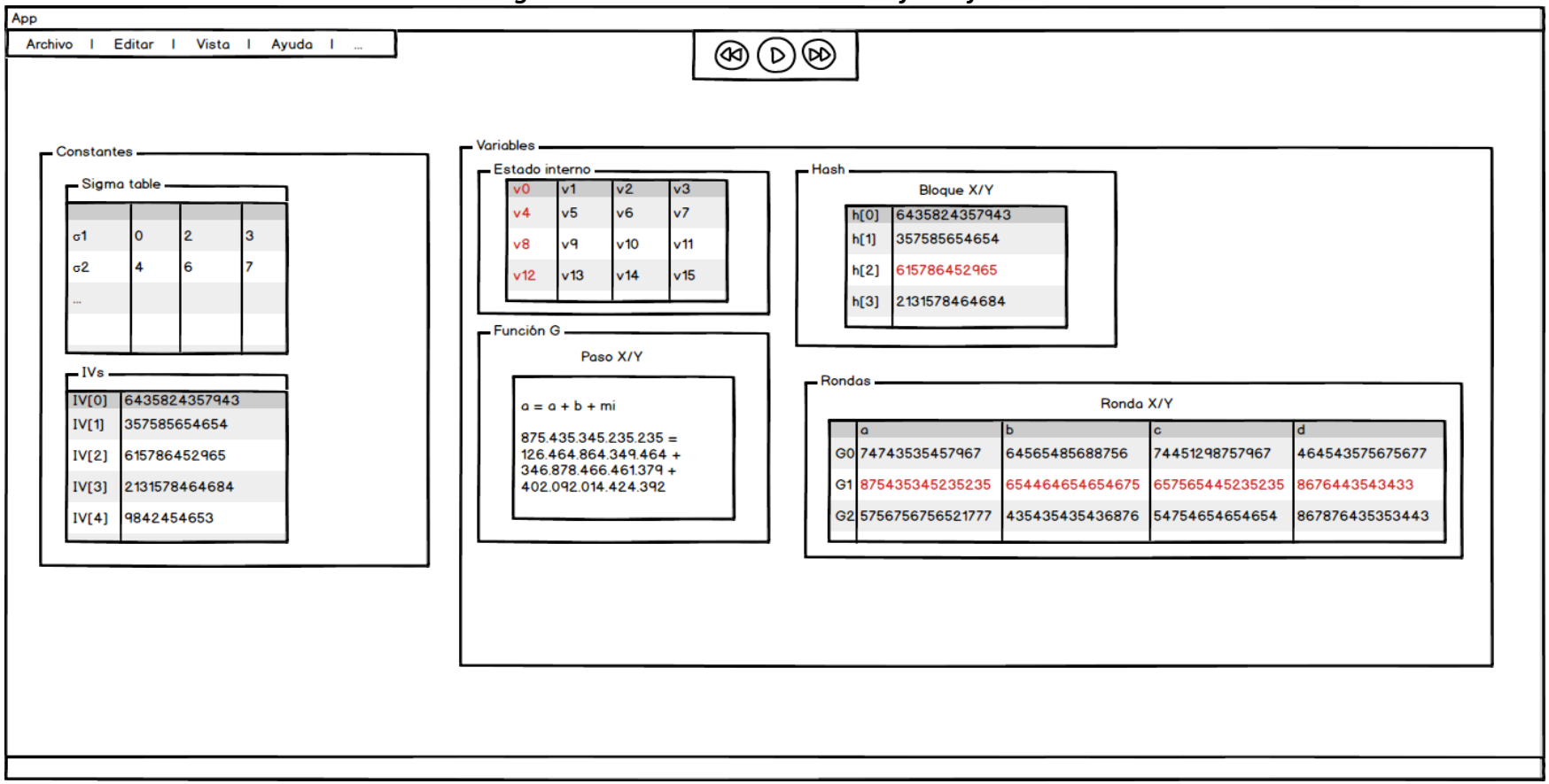


Fuente: Elaboración propia con Draw.io.

4.3. Boceto inicial

A continuación, en la **figura 26**, se muestra el boceto inicial para la interfaz del software.

Figura 26. Boceto inicial de la interfaz software.

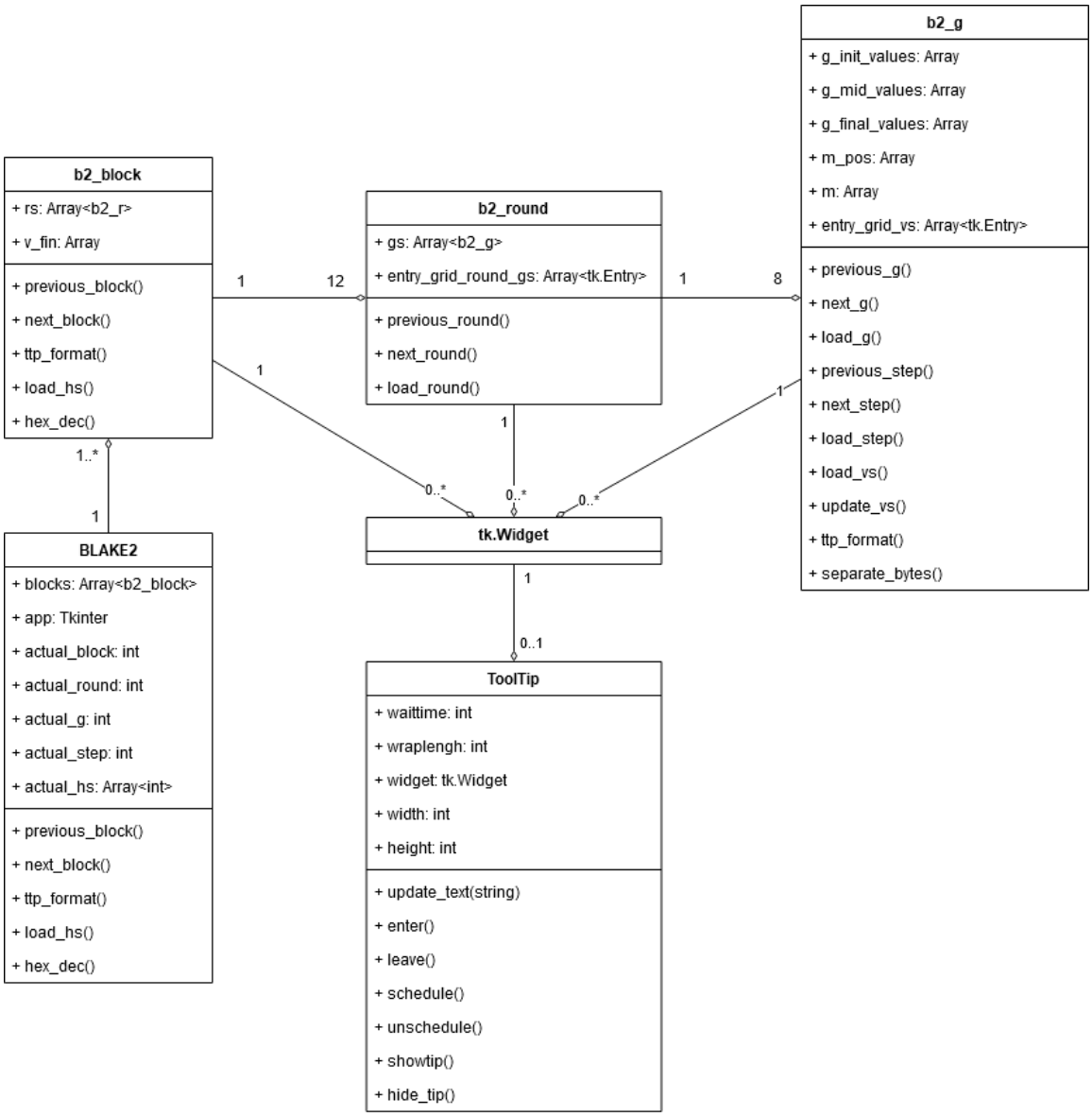


Fuente: Elaboración propia con Balsamiq Mockups.

4.4. Diagramas de clase

En la **figura 27** se muestra el diagrama de clases. No se incluyen los elementos relacionados con las interfaces ya que son demasiados y, además, no aportarían información relevante al diagrama de clases.

Figura 27. Diagrama de clases.



Fuente: Elaboración propia con Draw.io.

5. Resultados

Tras describir los objetivos, analizar el estado del arte, establecer la metodología de trabajo a seguir, la planificación y el marco tecnológico, en este capítulo se exponen los resultados obtenidos del desarrollo del presente TFM.

Esto incluye el proceso de desarrollo, la arquitectura del sistema, el aspecto visual final del mismo, testeo y problemas surgidos durante el desarrollo.

5.1. Arquitectura del sistema

La arquitectura del sistema desarrollado es monolítica, de forma con todos los elementos del sistema están integrados en el propio proyecto. La aplicación está estructurada con los siguientes directorios:

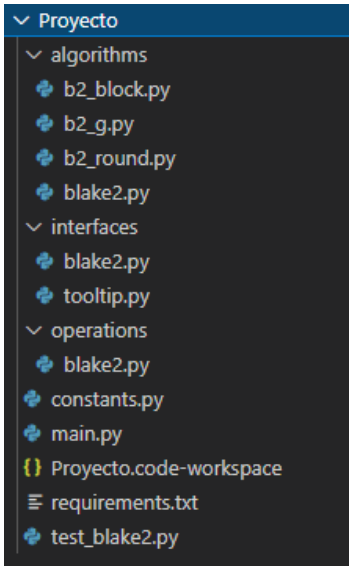
- **main.py**: es el archivo que permite inicializar la aplicación. Carga la ventana inicial del sistema y queda a la espera de que el usuario ejecute alguna acción.
- **constants.py**: contiene las constantes globales que se utilizan en el programa. Esto se ha realizado para que, en caso de tener que realizar alguna modificación de cualquiera de ellas, se puedan modificar todas sus ocurrencias a la vez.
- **test_blake2.py**: contiene los test que se realizan para comprobar el correcto funcionamiento del software.
- **operations/**: es el directorio que contiene los archivos con los cuales se carga una función cuando el usuario ya ha elegido una en el menú.
    - ***blake2.py***: es el único archivo que existe ya que es la única función que ha sido implementada. Contiene la clase BLAKE2 y se encarga de inicializar los elementos necesarios para realizar el algoritmo.

- **interfaces/**: es el directorio que contiene los archivos con los cuales se crean, inicializan y actualizan las interfaces de la función que el usuario ha elegido en el menú.
    - ***blake2.py****:* crea todos las ventanas y sus subelementos necesarios de la interfaz para mostrar los pasos: IVs, sigma, estado interno, funciones G, rondas, bloques y la ventana de ayuda.
    - ***tooltip.py****:* utilizando *tkinter,* permite crear ventanas emergentes para mostrar la información adicional que fuera necesaria en cada caso.
- **algorithms/**: es el directorio que contiene los archivos con la lógica necesaria para realizar la función hash y poder guardar la información de todos los pasos intermedios que posteriormente serán mostradas al usuario.
    - **blake2.py**: es una implementación de BLAKE2 en Python[6] y es la base sobre la que se ha realizado el software. Es más lenta que su equivalente en C, pero para el propósito educativo de este proyecto es suficiente. Se ha modificado la versión original (cuyo output era únicamente el hash final) para poder obtener la información es representada en las interfaces.
    - **b2_block**: contiene la clase que es una representación de cada uno de los bloques que se procesan durante el cálculo del hash.
    - **b2_round**: contiene la clase que es una representación de cada una de las rondas que se procesan durante el cálculo del hash.
    - **b2_g**: contiene la clase que es una representación de cada una de las funciones g (y sus pasos) que se procesan durante el cálculo del hash.

En la **figura 28** se puede ver la estructura de directorios del proyecto.

[6] https://github.com/buggywhip/blake2_py

Figura 28. Organización del proyecto.



Fuente: Elaboración propia.

5.2. Proceso de desarrollo

El desarrollo ha seguido una metodología basada en el proceso de *desarrollo iterativo e incremental*, de forma que se ha dividido en las diferentes iteraciones que serán detalladas en este apartado.

5.2.1. Iteración 0: Estudio del estado del arte, algoritmos y herramientas existentes.

En esta primera iteración se realizó un estudio del estado del arte para comprender cual es la situación actual en lo que respecta a los algoritmos de criptografía, centrando el foco en las funciones hash.

Por otro lado, se hizo un estudio de las diferentes herramientas existentes para estos algoritmos que tuvieran un enfoque educativo, permitiendo no únicamente hacer el cálculo correspondiente, sino ver cómo se ha realizado ese proceso paso a paso.

Los principales algoritmos que se analizaron fueron:

- **DES**: es un algoritmo de cifrado simétrico creado en los años 70. Se dejó de utilizar principalmente debido a que se encontraron debilidades en el diseño y también al pequeño tamaño de las claves.

  Se descartó esta opción ya que es un algoritmo bastante antiguo, que no se usa en la actualidad y para el cual ya existen una gran cantidad de herramientas (DESCalc, S-DES Calculator, Mbrown1413 DES, SafeDES, etc.).

- **AES**: es el algoritmo de cifrado simétrico sucesor de DES, usado todavía en la actualidad. Se creó en 2001, tras una competición en la que participaron quince diseños, siendo el ganador Rijndael. Uno de los elementos que más incrementó la seguridad en comparación con su predecesor es el incremento en el tamaño de las claves (pasando de 56 bits a 128, 192 o 256).

  Se descartó principalmente debido a que ya existe una gran cantidad de herramientas (AES-Cryptographic-Tool, CryptoLab, CrypTool, AESphere, etc.), por lo que no se encontró una forma de poder aportar valor.

- **Keccak**: en 2006, el NIST comenzó una competición (en la cual participaron más de 50 diseños) para crear un nuevo algoritmo de hash, SHA-3. En 2012 acabó, siendo el ganador Keccak. Utiliza una construcción de esponja, donde los datos son "absorbidos" y procesados para mostrar una salida de longitud deseada.

  SHA-3 es bastante usado en la actualidad y no existen tantas herramientas para él, Sin embargo, es necesario destacar CrypTool 2, la cual permite ver todos los pasos de las distintas fases del algoritmo con una representación visual que permite seguir el proceso de forma muy sencilla e intuitiva. Debido a esto, se decidió valorar otras opciones.

- **BLAKE**: al igual que Keccak, participó en la competición de SHA-3, llegando a la final. El principal motivo por el que no ganó fue porque se parecía mucho a SHA-2, y uno de los objetivos de la competición era tener una alternativa por si en algún momento fallaba SHA-2.

  Se valoró esta opción ya que no hay muchas herramientas educativas de él, siendo la más destacable CrypTool 2 (aunque a diferencia de con Keccak, en este casi no se muestra los detalles internos del proceso del algoritmo). Sin embargo, durante el estudio del mismo se descubrió BLAKE2, el sucesor de BLAKE, por lo que se valoró más esa opción.

- **BLAKE2**: ya se han comentado los detalles y origen del algoritmo en el capítulo 2.

5.2.2. Iteración 1: Elección del algoritmo y estudio y comprensión del mismo.

En esta iteración, partiendo del análisis realizado en la iteración anterior, se eligió finalmente BLAKE2. Los demás fueron descartados por los motivos ya comentados. Los principales motivos por los que se eligió BLAKE2 son:

- **Falta de herramientas**: como ya se comentó en el capítulo 2, hay pocas herramientas (al menos públicas) sobre BLAKE2. Además, ninguna permite ver cómo funciona la función paso por paso.

- **Es relativamente reciente**: ya que se creó en 2012 y su RFC en 2015 (Saarinen, 2015). Teniendo en cuenta el tiempo que tardan este tipo de procesos en estandarizarse, y en comparación con los otros algoritmos estudiados, es bastante reciente.

- **Es ampliamente usado**: una gran cantidad de bibliotecas y herramientas lo utilizan: OpenSSL, Crypto++, librsync, checksum, WinRAR, Sodium, Noise (usado en Whatsapp), el Kernel de Linux, Botan, WireWard, etc.

- **Es rápido**: en la **figura 14** se puede ver una comparativa con otros algoritmos. Esto se debe a optimizaciones en las rotaciones, uso de little endian, mejoras en el padding, realiza menos rondas, etc.

- **Es seguro**: tanto BLAKE como el resto de finalistas de la competición obtuvieron buenos resultados en las pruebas de seguridad que se realizaron en (Chang, 2012). Debido a que BLAKE2 parte de la base de BLAKE, su nivel de seguridad es similar. No se ha demostrado los cambios que se realizaron en BLAKE2 respecto a BLAKE reduzcan su nivel de seguridad. Además, en la actualidad no se conoce ningún ataque o debilidad eficiente (Aumasson, 2013).

Una vez elegido el algoritmo, se realizó un profundo estudio del funcionamiento interno del mismo en base a la documentación oficial de los creadores y analizando diferentes implementaciones existentes, tanto en Python como en C.

Este estudio permitió comprender al detalle el funcionamiento del algoritmo y poder realizar un diseño inicial para la representación de cada uno de los elementos que se usan internamente algoritmo y cómo estos van cambiando, de forma que sea lo más sencillo e intuitivo posible para la persona que lo esté estudiando.

5.2.3. Iteración 2: Elaboración del plan inicial con el alcance del proyecto, requisitos, planificación y desarrollo de bocetos y diagramas.

En esta iteración se especificaron los requisitos funcionales y no funcionales, junto a una breve descripción de los mismos. Seguido a esto se definieron los casos de uso, el respectivo diagrama de casos de uso, y se realizaron diagramas de secuencia de las principales funcionalidades del sistema y los diagramas de clases.

Todos estos elementos ya han sido desarrollados en el capítulo 4, por lo que no se volverán a comentar en esta sección.

5.2.4. Iteración 3: Elección del marco tecnológico, instalación y configuración de las herramientas a utilizar durante el desarrollo.

En esta iteración se realizó el estudio, elección, instalación y configuración de las herramientas que han sido utilizadas durante el desarrollo. Se comentará brevemente algunas de las opciones, tanto las que se descartaron como las que fueron elegidas finalmente, junto al motivo de la elección.

Como lenguaje de programación se eligió **Python 3.8**. Esto se debe principalmente a que es el lenguaje de programación con el que el autor tiene más experiencia, pero hay otros factores que influyeron en esta decisión: es muy flexible, ampliamente utilizado, tiene una gran cantidad de bibliotecas con funcionalidades matemáticas y de interfaces gráficas, etc.

La principal alternativa que se valoró fue **C**, pero fue descartado ya que el autor casi no tenía experiencia con él, lo cual ralentizaría considerablemente el desarrollo del proyecto. Además, teniendo en cuenta el alcance y características del proyecto, no son vitales algunas de las características más destacables de C, como es la velocidad.

Respecto a la biblioteca para crear las interfaces se eligió **Tkinter**. Los principales motivos de esta decisión son: el la más usada, es muy flexible, viene preinstalada con Python, tiene una baja curva de aprendizaje, es open source y es más indicada para proyectos pequeños (lo cual encaja bastante con este proyecto, teniendo en cuenta su alcance). Además de esto, se analizaron las características y capacidades de esta biblioteca para asegurar que con ella se podrían alcanzar todos los objetivos planteados.

Otras alternativas que se valoraron son: PyQT 5, PySide, Kivy y wxPython. Finalmente se eligió Tkinter por los motivos ya comentados.

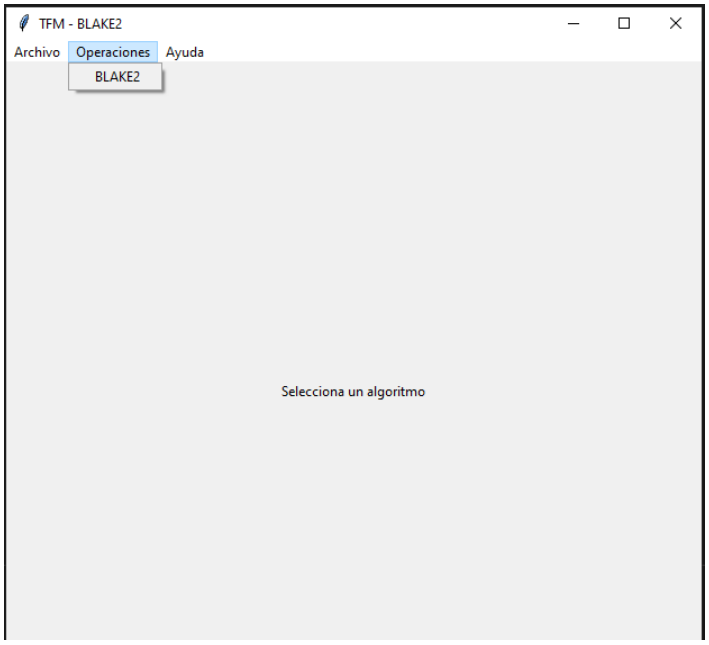
Respecto al resto de herramientas elegidas no se entrará en detalle ya que no son relevantes para el desarrollo del software.

Una vez se definió el marco tecnológico se hicieron algunas pruebas de concepto para entender al detalle cómo funciona Tkinter y cómo organiza y maneja las interfaces.

5.2.5. Iteración 4: Implementación de la interfaz inicial y funcionalidades básicas

En esta iteración se creó el proyecto en GitHub y, usando como base la prueba de concepto realizada en la fase anterior, se comenzó a realizar la estructura de ventanas del software. En la **figura 29** se puede ver el resultado. Esto se realizó como previsión por si en el futuro se añadían otras funciones o variaciones de BLAKE2.

Figura 29. Interfaz inicial del programa.



Fuente: Elaboración propia.

El código para la misma (perteneciente al archivo *main.py*) es sencillo, ya que solo se utilizan algunos de los elementos básicos de Tkinter. Por ello, solo se comentarán algunas de las líneas principales:

La primera linea crea una instancia de la clase Tk, la cual además crea el intérprete asociado a Tcl y la ventana principal de la aplicación. La segunda muestra todo en pantalla y se queda en espera hasta que el usuario interaccione con la aplicación o acabe el programa.

```
root = tk.Tk()
root.mainloop()
```

Otros de los elementos principales son el *Frame* (un contenedor para otros widgets), *Label (*un widget que contiene una cadena de texto estática) y *Menu* (una barra de menú).

```
frame = Frame(root)
Label(frame, text='Selecciona un algoritmo')
main_menu = Menu(root, tearoff=0)
```

Por otro lado, tenemos los métodos *title* (dar un nombre de la ventana), *geometry* (establecer el tamaño de la ventana), *grid* (posicionar widgets en un frame en una estructura cuadriculas) y *config* (cambiar atributos de un widget una vez creado, en este caso agregarle el menú creado anteriormente a la ventana principal.

```
root.title(title)
root.geometry(screen_size)
frame.grid()
root.config(menu = main_menu)
```

Por último, tenemos los métodos *add_cascade* (añadir un submenú a un menú) y *add_command* (añadir un comando a un submenú).

```
main_menu.add_cascade(label = "Operaciones", menu = operation_menu)
operation_menu.add_command(label = "BLAKE2", command = make_blake2)
```

En esta última, *command* hace referencia a la función que se llamará en caso de que el usuario haga click en un comando de un submenú. En este caso crea la función crea un objeto BLAKE2, el cual se comentará a continuación.

```
blake2 = BLAKE2(master = root, app=self)
```

Al crear esta clase se carga la interfaz para realizar la función BLAKE2, como se puede ver en la **figura 30**. Se utiliza el método *grid_forget* para vaciar el frame:

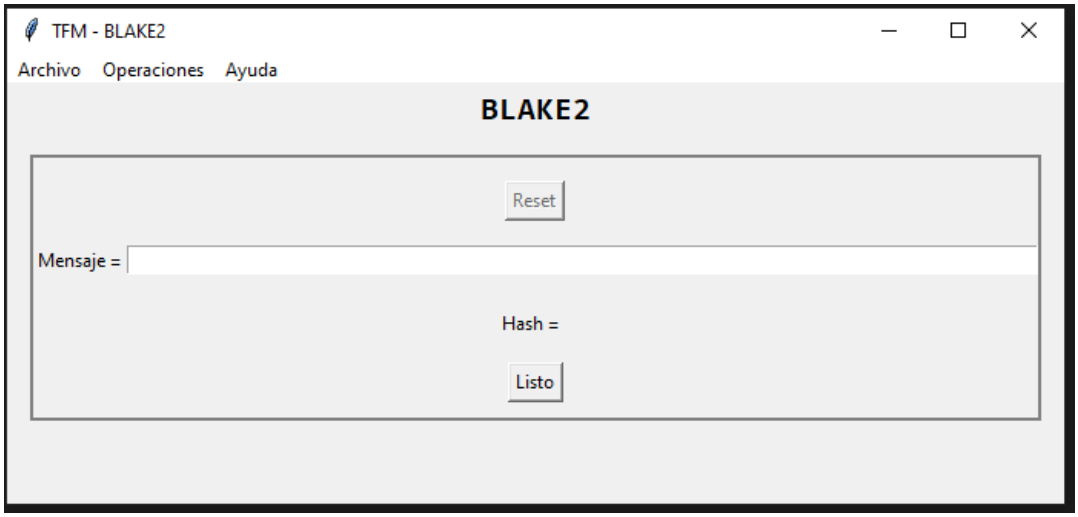
```
frame.grid_forget()
```

Y se crean los widgets necesarios. Aquí aparecen dos widgets nuevos: *Entry* (permite al usuario introducir una línea de texto) y *Button* (llama a una función cuando el usuario hace click en él).

```
message_entry = tk.Entry(message_frame, width=200)
calc_hash_button = tk.Button(calc_hash_frame)
calc_hash_button.config(text = "Listo", command = lambda: calc_hash())
```

La función *calc_hash* es la que realizará el cálculo del hash (guardando todos los valores intermedios necesarios) y llamará a la generación de las diferentes ventanas que mostrarán los datos del proceso. Estos se implementan en las siguientes iteraciones.

Figura 30. Interfaz de BLAKE2.

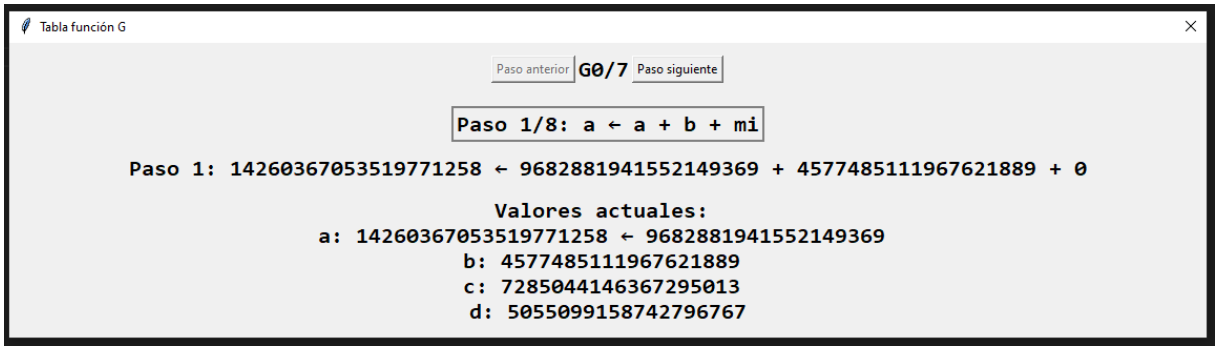


Fuente: Elaboración propia.

5.2.6. Iteración 5: Visualización de todos los pasos de una función G

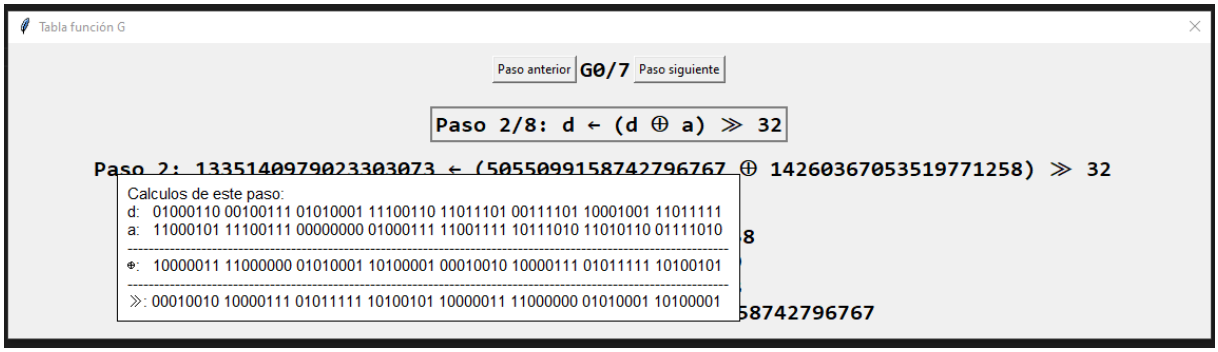
En esta iteración se ha implementado la lógica necesaria para guardar los valores de cada uno de los pasos de una función G, además de crear las ventanas que mostrarán esta información. Esta misma ventana permitirá también desplazarse a través de las funciones G (se implementará en la iteración 6). En la **figura 31 y la figura 32** se puede ver el resultado final.

Figura 31. Ventana de pasos.



Fuente: Elaboración propia.

Figura 32. Ventana de pasos con ToolTip.



Fuente: Elaboración propia.

Para realizar esto tenemos la clase *b2_g* (ver diagrama de clases). En una función G tenemos diferentes valores. Primero a, b, c y d, los cuales se van modificando a lo largo de los pasos de la función G, tomando tres valores distintos, que han sido definidos como *init*, *mid* y *final*:

```
g_init_values = [0]*4
g_mid_values = [0]*4
g_final_values = [0]*4
```

También tenemos *m*, que son las palabras del mensaje a utilizar (m_i y m_j), y *m_pos*, que es su posición (fila y columna) en la tabla sigma:

```
m_pos = [0]*2
m = [0]*2
```

Dentro del código de la función hash se han ido guardando los valores de G según se iban calculando (en *init*, *mid* o *final* según el paso) y los valores mi de *m*.

```
g_init_values[0] = va
g_init_values[1] = vb
g_init_values[2] = vc
g_init_values[3] = vd
m[0] = msri2
m[1] = msri21
```

Ya que todavía no están implementadas las demás estructuras (rondas, bloques, etc.), solo se han guardado los datos de la última función G que ha ejecutado el algoritmo.

La función principal para cargar estos datos es *load_step* la cual, en función del paso en el que se esté, carga los valores necesarios. A continuación, se muestra un fragmento de código (simplificado, para facilitar la lectura) de los dos primeros pasos:

```
if actual_step == 1:
    step_val.set("Paso 1/8: a ← a + b + mi")

    step_result_val.set("Paso 1: " + g_mid_values[0] + " ← " +
    g_init_values[0] + " + " + g_init_values[1] + " + " + m[0]

    gs_act_vals.set("Valores actuales: \na: " + g_mid_values[0] + " ← " +
    g_init_values[0] + " \nb: " + g_init_values [1] + " \nc: " +
    g_mid_values[2] + " \nd: " + g_init_values[3])

elif actual_step == 2:
    step_val.set("Paso 2/8: d ← (d ⊕ a) ≫ 32")

    step_result_val.set("Paso 2: " + g_mid_values[3] + " ← (" +
    g_init_values[3] + " ⊕ " + g_mid_values[0] + ") ≫ 32 ")

    gs_act_vals.set("Valores actuales: \na: " + g_mid_values[0] + "\nb: " +
    g_init_values[1] + " \nc: " + g_init_values[2] + " \nd: " +
    g_mid_values[3] + "←" + g_init_values[3])
```

Los botones “Paso anterior” y “Paso siguiente” llaman respectivamente a las funciones *previous_step* y *next_step*, que hacen algunas comprobaciones y posteriormente llaman a la ya comentada *load_step.*

Los pasos 1, 3, 5 y 7 de una función son sumas, por lo que no se entra en detalle en la explicación del paso. Sin embargo, los pasos 2, 4, 6 y 8 tienen operaciones xor y desplazamientos por lo que, para que sea más fácil para el usuario entender que está ocurriendo, se optó por crear la clase *ToolTip*. Esta clase, como se puede ver en la **figura 32**, hace que al mantener el cursor sobre una operación se muestren las operaciones realizadas.

Esta clase crea una ventana nueva con una *Label* (con la cadena de texto que se le pase como parámetro) y elimina el resto de la ventana, dejando solo la *Label*.

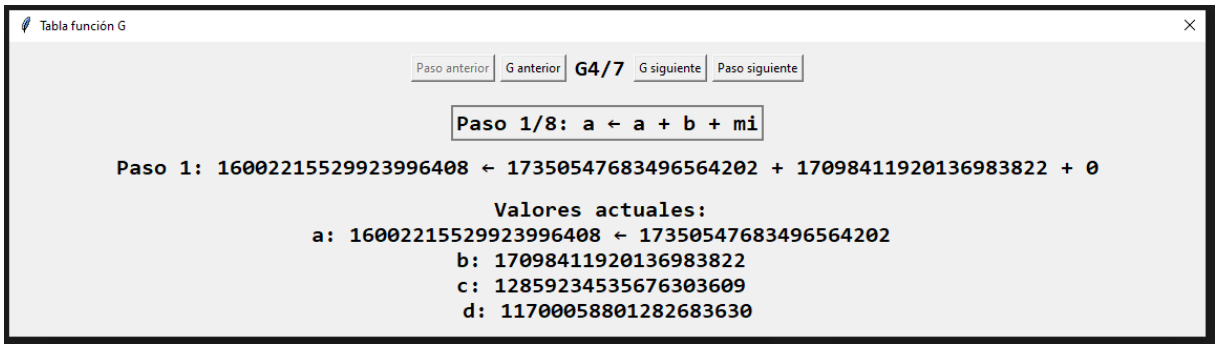
Para la representación de los números en binario se creó la función *separate_bytes*, la cual recibe un int, lo convierte a binario, añade ceros hasta tener 64 dígitos si es necesario y por último los separa en grupos de ocho para facilitar la lectura.

```
def separate_bytes(data):
    result = bin(data)[2:]
    if len(result) < 64:
        result = ("0" * (64 - len(result))) + result
    result = " ".join([result[i:i+8] for i in range(0, len(result), 8)])
    return result
```

5.2.7. Iteración 6: Visualización de todas las funciones de una ronda

En esta iteración se implementaron métodos para el desplazamiento en las funciones de una ronda y se adaptó la ventana creada en la iteración anterior para que muestre también el resto de funciones, como se ve en la **figura 33,** donde el usuario está situado en la función 4.

Figura 33. Ventana de funciones G.



Fuente: Elaboración propia.

La forma en la que se ha implementado esto es similar a como se hizo con los pasos: tenemos la función *load_g* y las funciones *previous_g* y *next_g*, a las cuales se las llama cuando se hace click en los respectivos botones.

previous_g y *next_g* hacen algunas comprobaciones y llaman a *load_g*, la cual hace algunos cambios en la interfaz y llama a *load_step* para que cargue los valores del primer paso de la función g que corresponda. Las líneas principales son:

```
load_g(actual_g)
gs_val_str.set("G" + actual_g + "/" + cons.n_gs - 1)
load_step(gs[actual_g])
```

gs es una lista con los datos de todas las funciones G de una ronda (de forma similar a en la iteración anterior, guarda solo los de la última que se ejecuta).

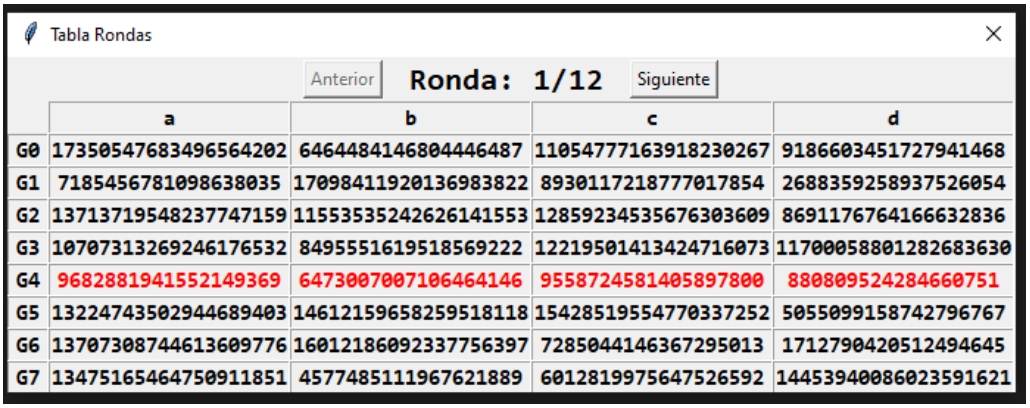
Para realizar esto, en la ejecución del algoritmo se modificó la forma en la que se guardaban los valores en la iteración 5, de forma que se guardan en una lista, donde la posición es el número de la función:

```
gs[ng].g_init_values[0] = va
gs[ng].g_init_values[1] = vb
gs[ng].g_init_values[2] = vc
gs[ng].g_init_values[3] = vd
gs[ng].m[0] = msri2
gs[ng].m[1] = msri21
```

5.2.8. Iteración 7: Visualización de todas las rondas de un bloque

En esta iteración se creó la clase *b2_round*, que es la responsable de la lógica de cada una de las rondas que se realiza dentro de un bloque, y se creó la ventana para el desplazamiento entre las rondas. La interfaz final se puede ver en la **figura 34**.

Figura 34. Ventana de rondas.



	a	b	c	d
G0	17350547683496564202	6464484146804446487	11054777163918230267	9186603451727941468
G1	7185456781098638035	17098411920136983822	8930117218777017854	2688359258937526054
G2	13713719548237747159	11553535242626141553	12859234535676303609	8691176764166632836
G3	10707313269246176532	8495551619518569222	12219501413424716073	11700058801282683630
G4	9682881941552149369	6473007007106464146	9558724581405897800	880809524284660751
G5	13224743502944689403	14612159658259518118	15428519554770337252	5055099158742796767
G6	13707308744613609776	16012186092337756397	7285044146367295013	1712790420512494645
G7	13475165464750911851	4577485111967621889	6012819975647526592	14453940086023591621

Fuente: Elaboración propia.

En esta clase el atributo más importante es *gs*, que es una lista de objetos *b2_g*, es decir, de las funciones G. De esta forma, los valores guardados en la iteración anterior son asignados a cada objeto *b2_g* correspondiente.

```
gs = [b2_g() for x in range(cons.n_gs)]
```

Para guardar los datos de cada ronda se modificó la forma en la que se guardaban los valores en la ejecución del algoritmo de forma que se guardan en una lista, donde la posición es el número de la ronda:

```
rs.[nr].gs[ng].g_init_values[0] = va
rs.[nr].gs[ng].g_init_values[1] = vb
rs.[nr].gs[ng].g_init_values[2] = vc
rs.[nr].gs[ng].g_init_values[3] = vd
rs.[nr].gs[ng].m[0] = msri2
rs.[nr].gs[ng].m[1] = msri21
```

Igual que en los casos anteriores, tenemos la función *load_round* y las funciones *previous_round* y *next_round*, a las cuales se las llama cuando se hace click en los respectivos botones.

previous_round y *next_round* hacen algunas comprobaciones y llaman a *load_round* y *load_g*.

load_round hace algunos cambios en la interfaz y carga los valores de todas las a, b, c y d que son obtenidas al acabar la ejecución de cada una de las funciones G de la ronda.

También se llama a *load_g* para que cargue los datos de la primera función G (y en consecuencia, también de su primer paso) en la ventana de funciones G. Esto se hace para que cada vez que se cambie de ronda el software se desplace al inicio de la misma.

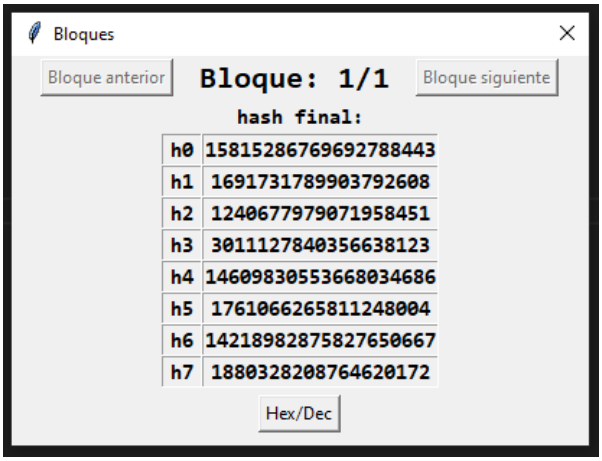
```
load_round(rounds = rs, n_round = actual_round)
```

Además, como se puede ver en la **figura 34,** se marcan en color rojo los valores que están siendo calculados en este momento para facilitar el seguimiento del proceso.

5.2.9. Iteración 8: Visualización de todos los bloques de la función

En esta iteración se creó la clase *b2_block*, que es la responsable de la lógica de cada uno de los bloques que se calculan, y se creó la ventana para el desplazamiento entre bloques. La interfaz final se puede ver en la **figura 35**.

Figura 35. Ventana de bloques en decimal y hexadecimal.



Fuente: Elaboración propia.

En esta clase el atributo principal es *rs*, que es una lista de objetos *b2_round*, es decir, de las rondas de ese bloque (teniendo cada uno de los elementos de esa lista, su correspondiente lista de *gs*):

```
rs = [b2_round() for x in range(cons.n_rounds)]
```

Para guardar los datos de cada bloque se modificó la forma en la que se guardaban los valores en la ejecución del algoritmo de forma que se guardan en una lista, donde la posición es el número del bloque:

```
blocks[nb].rs.[nr].gs[ng].g_init_values[0] = va
blocks[nb].rs.[nr].gs[ng].g_init_values[1] = vb
blocks[nb].rs.[nr].gs[ng].g_init_values[2] = vc
blocks[nb].rs.[nr].gs[ng].g_init_values[3] = vd
blocks[nb].rs.[nr].gs[ng].m[0] = msri2
blocks[nb].rs.[nr].gs[ng].m[1] = msri21
```

Igual que en los casos anteriores, tenemos la función *load_hs* y las funciones *previous_block* y *next_block*, a las cuales se las llama cuando se hace click en los respectivos botones.

previous_block y *next_block* hacen algunas comprobaciones y llaman a *load_round*, *load_g* y *load_hs*. Esta última hace algunos cambios en la interfaz y carga las hs del bloque. Las hs son calculadas al finalizar todas las rondas de un bloque y después son guardadas en una lista.

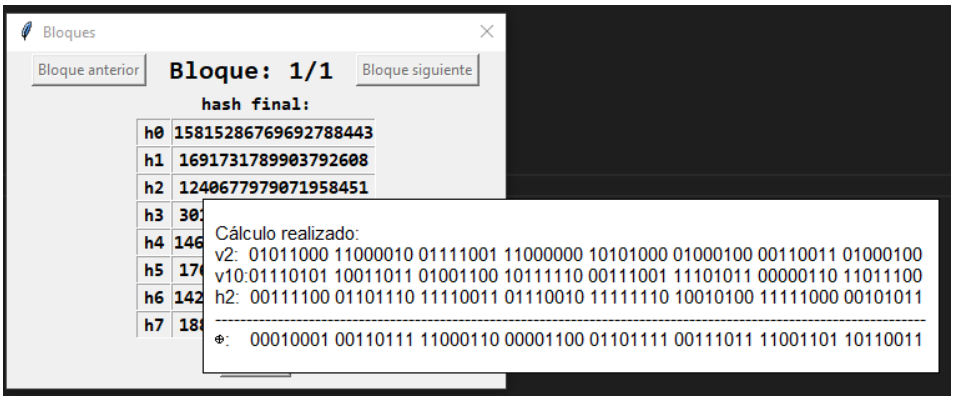
```
h = [h[i] ^ v[i] ^ v[i+8] for i in range(8)]
hs.append(self.h)
```

Para finalmente en *load_hs* cargar en la interfaz los valores de las hs de ese bloque.

```
load_hs(hs = hs[actual_block], n_block = actual_block)
```

Ya que para el cálculo de las hs se realizan operaciones xor, se añadieron elementos *Tooltip* que muestren el cálculo realizado en cada uno de ellos, como se puede ver en la **figura 36**.

Figura 36. Ventana de bloques con Tooltip.



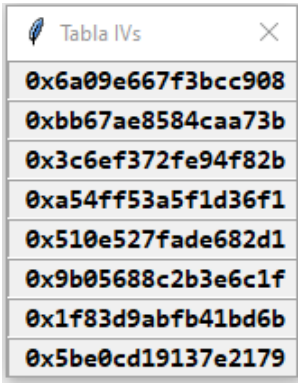
Fuente: Elaboración propia.

Por último, se creó la función *hex_dec* para permitir alternar entre la posibilidad de mostrar los valores de las hs entre hexadecimal y decimal. Esto se ha hecho así ya que las hs calculadas en el último bloque son, en hexadecimal, el hash final.

5.2.10. Iteración 9: Ventanas de sigma, estado interno e IVs

En esta iteración se crearon ventanas para mostrar la información de otros elementos que intervienen en el cálculo del hash: la tabla sigma, la tabla de estado interno y la tabla de IVs. Además, se añadió una guía de uso. En la **figura 37** se puede ver el resultado final de la interfaz.

Figura 37. Ventana de IVs



Fuente: Elaboración propia.

Para obtener los IVs se realizan una serie de cálculos ya explicados en el capítulo 2. En la ventana que se ha añadido solo se muestran sus valores, sin entrar en detalle en los cálculos. Esto se ha decidido así por varios motivos.

1. El cálculo de los IVs no es parte del proceso del hash, es un cálculo anterior a la ejecución del mismo.
2. Debido a que los IVs son siempre los mismos, en la práctica estos no se calculan, sino que se utilizan directamente.

La siguiente ventana que se creó fue la ventana de estado interno. Para su creación se implementaron dos funciones en la clase b2_g: *load_vs* y *update_vs*.

La primera es la que se utiliza cuando se inicializa la ventana, momento en el que carga en la tabla los valores iniciales, es decir, los que se obtienen al comenzar a operar con el primer bloque.

La segunda se utiliza para actualizar los valores, los cuales son mostrados en un *Tooltip* como se puede ver en la **figura 38**. Durante cada función G se calculan los valores de a, b, c y d (v1, v6, v11 y v12 en el caso de la **figura 38**) a lo largo de los distintos pasos de la función. Esta función es llamada tras ejecutar cada paso, actualizando el texto de los *Tooltip*. Se ha tenido que hacer una ligera distinción entre los primeros cuatro pasos (en vertical) y los últimos cuatro (en diagonal).

```
def update_vs(self, ng, list):
   if actual_g < 4:
      for pos in range(4):
         entry_grid_vs_ttp[ng][pos].update_text(list[pos])

   if actual_g >= 4:
      for pos in range(4):
         entry_grid_vs_ttp [(ng + pos) % 4][pos].update_text(list[pos])
```

Figura 38. Ventana de estado interno

Fuente: Elaboración propia.

La siguiente ventana creada es la ventana de sigma. Los valores que aparecen en la ventana son los mismos que en la tabla sigma (de forma similar a con los IVs, no se calcula, sino que se utiliza directamente). De forma similar a con la tabla de estado interno, si se mantiene el cursor por encima de un elemento de la tabla se mostrará el valor de la m correspondiente.

Puesto que los valores *m* y *m_pos* ya se estaban guardando, no se ha tenido que hacer ningún cambio en el procesamiento de los datos durante la ejecución de la función.

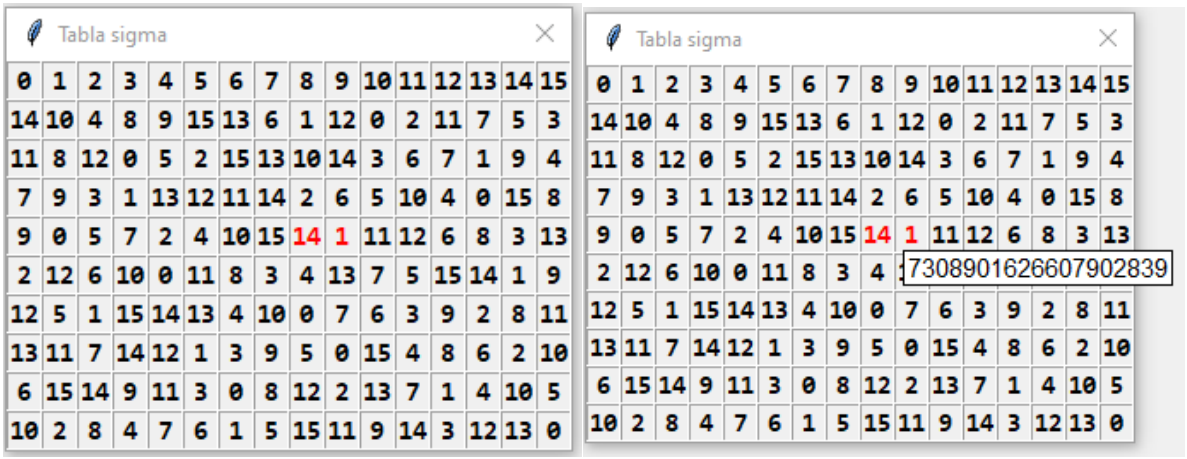
Para mostrar los datos se han implementado las funciones *update_sigma_window* y *update_sigma_color* en la clase *b2_block*, ya que *m* es una parte del mensaje, es decir, sus valores cambian cada vez que se procesa un bloque.

Por un lado, tenemos la función *update_sigma_window*, la cual actualiza los valores de las *Tooltip* de cada elemento de la tabla con los correspondientes a la *m* del bloque actual.

Por otro lado, *update_sigma_color se* encarga únicamente de marcar en rojo las *m* que se están utilizando en cada momento.

El resultado se puede ver en la **figura 39.**

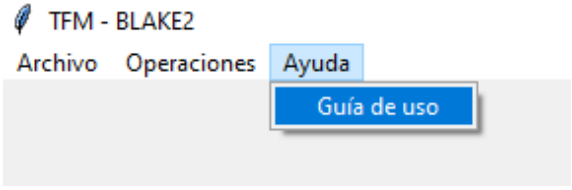
Figura 39. Ventana de sigma



Fuente: Elaboración propia.

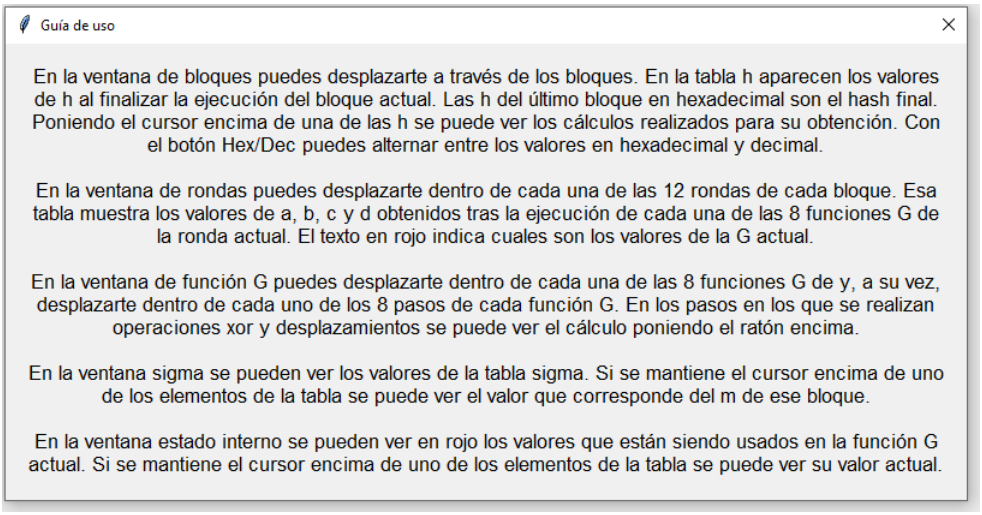
Por último, se creó guía de uso añadiendo un comando en el menú de ayuda, el cual abre una ventana nueva con información sobre cómo utilizar la herramienta. Estos elementos pueden verse en la **figura 40** y la **figura 41**.

Figura 40. Abrir guía de uso



Fuente: Elaboración propia.

Figura 41. Guía de uso



Fuente: Elaboración propia.

5.3. Testeo de la aplicación

Con el objetivo de comprobar el correcto funcionamiento del software desarrollado se decidió implementar una serie de tests, en el archivo *test_blake2.py*.

Finalmente, solo se implementó un test, el cual permite comprobar, contra otra biblioteca de hashes, que el hash final generado por la herramienta es el correcto.

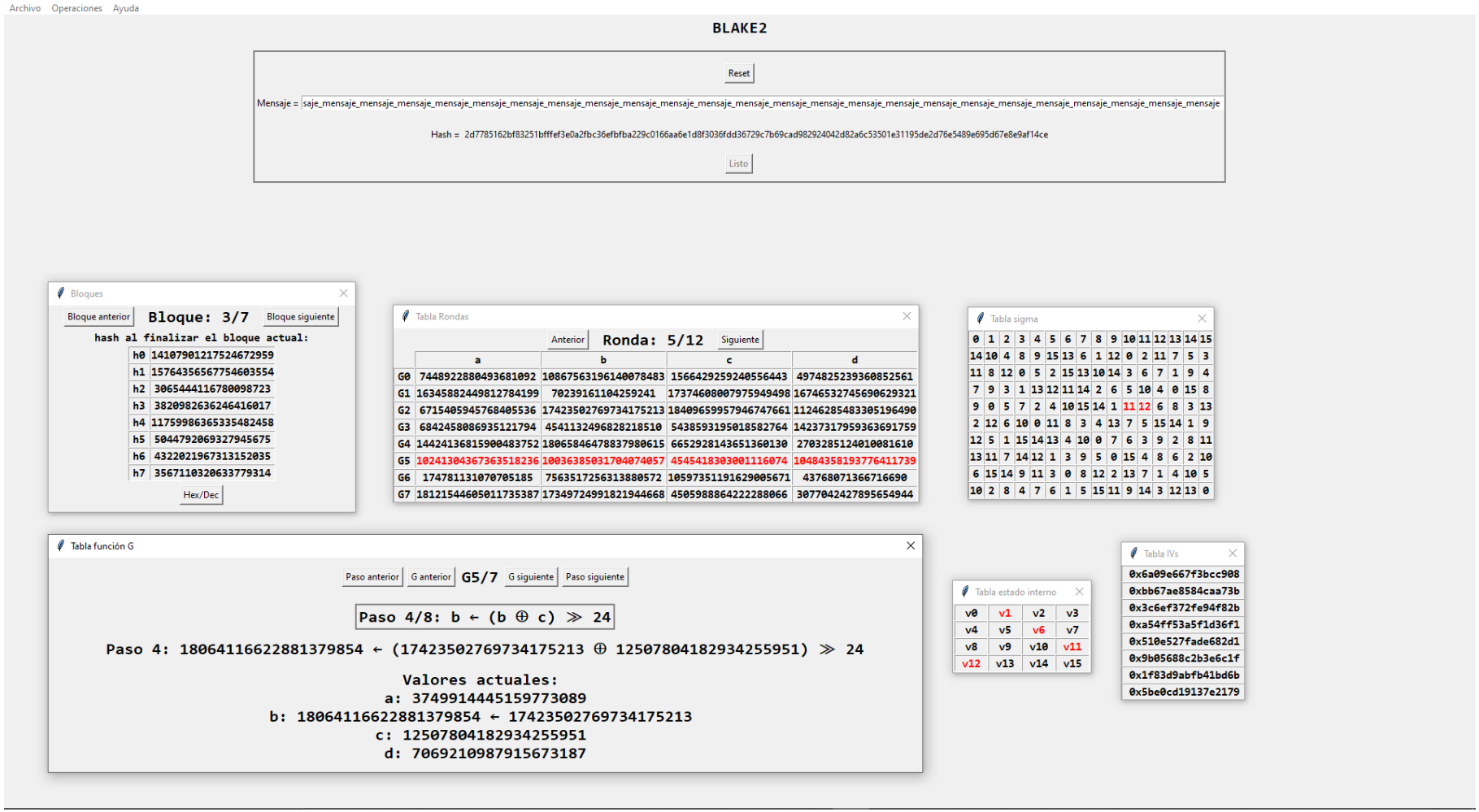
Solo se implementó este test debido a que no existe ninguna otra herramienta que utilice los valores intermedios que se calculan en el hash, haciendo que sea imposible implementar test sobre los cálculos de los mismos. A continuación, se muestra el código del test:

```
    def test_hashes_are_equal(self, value):
        # Hash with oficial library
        original_hash = hashlib.blake2b(value.encode('utf8')).hexdigest()

        #Hash with own implementation
        self.b2 = b2.BLAKE2b(digest_size=64, debug = True)
        self.b2.update(value.encode('utf8'))
        digest = self.b2.final()
        own_hash = binascii.hexlify(digest).decode()

        self.assertEqual(original_hash, own_hash)
```

5.4. Interfaz de usuario final

A continuación, en la **figura 42**, se muestra la interfaz de usuario final.

Figura 42. Interfaz de usuario final



Fuente: Elaboración propia.

5.5. Problemas surgidos durante el desarrollo

En este apartado se comentarán los principales problemas surgidos durante el desarrollo y que han limitado o dificultado su desarrollo.

- **Falta de capacidad de testeo.** Debido a que la herramienta desarrollada es la única que entra en detalles sobre los procesos internos del algoritmo, guardando sus valores intermedios, no se han podido realizar test unitarios para comprobar que los valores que se muestran son los correctos.

  Esto, sumado a la complejidad del algoritmo y a que los números que se calculan son muy grandes y cambian constantemente, han ralentizado considerablemente el desarrollo del TFM.

6. Conclusiones

En este capítulo se comentarán las conclusiones obtenidas tras el desarrollo del TFG, los objetivos que se han alcanzado y las posibles mejoras a realizar en sus próximas versiones.

6.1. Objetivos alcanzados

- **Objetivo: estudio el algoritmo BLAKE2.**

  Resultado: se ha estudiado y comprendido en detalle el algoritmo BLAKE2, estudiando de forma teórica la documentación oficial proporcionada por los autores, su RFC y analizando su funcionamiento en vivo con el debugger de Visual Studio Code.

- **Objetivo: análisis y comparativa de las herramientas criptográficas existentes.**

  Resultado: se han estudiado todas las herramientas que ha sido posible encontrar, permitiendo determinar las carencias que tienen en términos educativos y cómo sería posible mejorarlas.

- **Objetivo: análisis y valoración de posibles alternativas para desarrollar el software.**

  Resultado: se ha realizado un estudio de cómo están desarrolladas otras herramientas similares para poder establecer un marco tecnológico que permitiera cumplir con los objetivos y obtener una herramienta los más completa posible.

- **Objetivo: elección de una metodología de trabajo.**

  Resultado: se han estudiado diferentes metodologías de trabajo que, teniendo en cuentas las características y requisitos del software a desarrollar, permitiera trabajar de la forma más eficiente y organizada posible.

- **Objetivo: diseño de una interfaz de usuario.**

  Resultado: el análisis de otras herramientas similares ha permitido pulir el diseño de la interfaz de usuario para que esa sea lo más completa posible, a la vez que sencilla de utilizar y entender.

6.2. Trabajo futuro

Existen una serie de modificaciones y mejoras que se podrían realizar al software desarrollado para aportar más valor:

- Actualmente solo se puede calcular el hash del texto que se introduzca por teclado. Se podría añadir una funcionalidad para aplicarlo a un archivo seleccionado por el usuario.
- El hash se calcula con la función BLAKE2b-512. Se podría añadir el cálculo de otras versiones de BLAKE2 como BLAKE2b-256, BLAKE2b-384, BLAKE2s-128 o BLAKE2s-256.
- El hash se calcula sin tener en cuenta la posibilidad que ofrece BLAKE2 de utilizar una clave al calcular el hash. También existe el modo árbol, el cual no se utiliza en el software desarrollado y sería una característica deseable para añadir.
- Añadir la funcionalidad de realizar criptoanálisis para BLAKE2, además de realizar ataques a hashes, por ejemplo, por fuerza bruta.
- Permitir al usuario exportar los datos obtenidos del hash calculado en un formato como JSON o XML.

Referencias bibliográficas

Abood, O. G. (2018). *A survey on cryptography algorithms.* International Journal of Scientific and Research Publications, 8(7), 495-516.

Aumasson, J. P.-O. (2013). *BLAKE2: simpler, smaller, fast as MD5.* In International Conference on Applied Cryptography and Network Security (pp. 119-135). Springer, Berlin, Heidelberg.

Chang, S. J. (2012). *Third-Round Report of the SHA-3.* NIST Interagency Report, 7896, 121.

Evans, M. M. (2016). *Human behaviour as an aspect of cybersecurity assurance.* Security and Communication Networks, 9(17), 4667-4679.

Fowler, M. &. (2001). *The agile manifesto. Software development, 9(8), 28-35.*

Gençoğlu, M. T. (2019). *Importance of Cryptography in Information Security.* IOSR J. Comput. Eng, 21(1), 65-68.

Larman, C. &. (2003). *Iterative and incremental developments. a brief history. Computer, 36(6), 47-56.*

Ministerio del interior. (2021). Obtenido de http://www.interior.gob.es/documents/10180/11389243/Estudio+sobre+la+Cibercriminalidad+en+Espa%C3%B1a+2020.pdf/ed85b525-e67d-4058-9957-ea99ca9813c3

NIST. (2009). *Cryptographic Hash Algorithm Competition*. Obtenido de https://www.nist.gov/programs-projects/cryptographic-hash-algorithm-competition

Pittalia, P. P. (2019). *A comparative study of hash algorithms in cryptography. .* International Journal of Computer Science and Mobile Computing, 8(6), 147-152.

PWC. (2014). *US cybercrime: Rising Risks, Reduced Readiness. Key findings from the 2014 US State of Cybercrime Survey*. Obtenido de http://www.pwc.com/us/en/increasing-it-effectiveness/publications/assets/2014-us-state-of-cybercrime.pdf

Saarinen, M. J. (2015). *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC): IETF RFC 7693.*

Yildirim, E. (2016). *The importance of information security awareness for the success of business enterprises.* Advances in human factors in cybersecurity (pp. 211-222). Springer, Cham.

Anexo A. Código fuente

El código fuente del proyecto está disponible en un repositorio público en GitHub:

https://github.com/ismaelvr/TFM

Anexo B. Guía de instalación

1. Descargar el código fuente del repositorio indicado en el Anexo A.
2. Verificar que se tiene instalado *Python 3.8.X*. En caso contrario, instalarlo.
3. Verificar que se tiene instalado *pip*. En caso contrario, instalarlo.
4. Instalar las dependencias con el comando *pip install -r requirements.txt*
5. Ejecutar la aplicación con el comando *python main.py*