Tabla de Contenido

1. Introducción 2
2. Problema principal (Incluye posibles ataques) 2
3. Generación de llaves secretas 3
4. Uso de PSK para asegurar el entorno 4
4.1 Motivación de uso de las PSK 4
4.2 ¿Que son las PSK? 4
4.3 Modos de compartición de las PSK 5
4.4 Autenticación y compartición de llaves con PSK 5
5. Estrategia de cifrado 8
5.1 Optimización y uso de memoria del encriptado con threads 9
6. Estrategia de descifrado 10
7. Estrategia para uso de llaves dinámicas 10
8. Las correcciones más significativas 11
9. Librerías utilizadas 11
10. Estrategia de verificación y validación usada para medir la calidad interna del código 12
10.1 Pruebas de rendimiento: por threads 12
10.1.1 100MB 12
10.1.2 500MB 12
10.1.3 1GB 12
10.1.4 5GB 12
10.1.5 10GB 13
10.2 Verificación calidad interna del código 13
11. Referencias 14

PowerCoders

Autores:

Diego Alejandro Pulido

Danny Camilo Muñoz Sanabria

Johan Bautista

Juan Camilo López

Solución en seguridad y eficiencia para comunicaciones satelitales

1. Introducción

Las soluciones de criptografía generalmente toman bastantes recursos computacionales para llegar a cumplir el objetivo brindar comunicaciones seguras. En general, los métodos de comunicación usan certificados y algoritmos de encriptación asimétricos que son bastante costosos para asegurar la comunicación entre dispositivos (Véase el funcionamiento de TLS1.2). Sin embargo, no todos los dispositivos tienen mucho poder de cómputo para asegurar comunicaciones seguras, por lo que se necesitan otras estrategias óptimas para asegurar comunicaciones en dispositivos con pocos recursos. Por esta razón, en este documento explicamos nuestra propuesta de solución que brinda comunicación segura para dispositivos de bajo poder de cómputo como los satélites, usando algoritmos criptográficos eficientes como AES256/GCM y estrategias como el uso de PSK(Pre-Shared-Keys) para autenticación de los dispositivos.

2. Problema principal (Incluye posibles ataques)

Antes de empezar con la solución, es necesario repasar los posibles problemas que fueron pieza importante para el diseño de nuestra solución. El primer problema es bastante claro, y corresponde al límite computacional propuesto por el reto, de este principal problema subyacen los demás. Entre ellos se encuentran:

1. ¿Como podíamos generar la llave secreta en una sola comunicación?
    a. No podemos usar Diffie-Hellman!
2. ¿Como hacíamos que el intercambio de la llave secreta se compartiera en un entorno inseguro (evitar Man-In-The-Middle)?
3. ¿Como podemos evitar uso de certificados digitales con llaves asimétricas?
4. ¿Como optimizamos el encriptado y desencriptado a pesar de usar AES256/GCM?

Estos problemas los resuelve nuestra solución a la perfección, y a lo largo de este documento vamos a explicar a detalle la solución de cada una.

3. Generación de llaves secretas

Teniendo en cuenta las restricciones de comunicación, queda totalmente descartado el uso de Diffie-Hellman para solucionar la generación de llaves secretas. Por esta razón, creamos una nueva manera de generar las llaves basándonos en una combinación entre usar RSA y PBKDF2 (Password-Based Key Derivation Function).

Para comenzar la explicación de la generación de llaves, tenemos 3 variables importantes que componen la generación de las llaves, que son:

- **Pre-Seed:** Es una semilla pre guardada en ambos extremos de la comunicación, al igual de cómo funcionan las PSK. Esta semilla, es un texto al azar estático, que se usa como contraseña para poder usar la función PBKDF2. Al ser pre compartida, no se comparte durante la comunicación, y nadie más puede saberla a no ser que vulneren físicamente el satélite o el dispositivo en la base terrestre que mantiene la comunicación.
- **Numero primo (P):** Este parámetro, es un numero primo aleatorio generado físicamente por ambos nodos de la comunicación. Al igual que la Pre-Seed, es un atributo estático y pre-guardado en el satélite y en la base que nadie más puede conocer y que no se comparten por medio de la comunicación.
- **Numero Random (R):** Este parámetro, es un numero generado aleatoriamente por el satélite antes de enviar cada imagen. Este número, es el que nos brinda más entropía a nuestra futura clave, ya que al ser parte de la contraseña, hace que cada clave generada para cada imagen siempre sea diferente y aleatoria, asegurando el requerimiento de las claves dinámicas.

Explicados estos parámetros se usan en la función PBKDF2 que tiene dos parámetros para iniciar la clave, que son la contraseña y una Salt. En este caso, para suplir el parámetro de la contraseña, usamos la Pre-Seed, y para la Salt usamos el numero random R.

Decidimos usar PBKDF2, ya que permite la creación de llaves seguras a partir de semillas. Además, este algoritmo ofrece una capa de más de seguridad, ya que al hacer uso de un salt, protege a la llave en contra de “Rainbow Table Attacks” en caso de que la semilla guardada sea vulnerable. Por otro lado, este algoritmo es customizable en cuanto a los ciclos que tiene que hacer para generar la llave, lo que permite configurarlo para que funcione en equipos de bajo poder computacional.

En general, la generación de la llave se ve la siguiente manera:

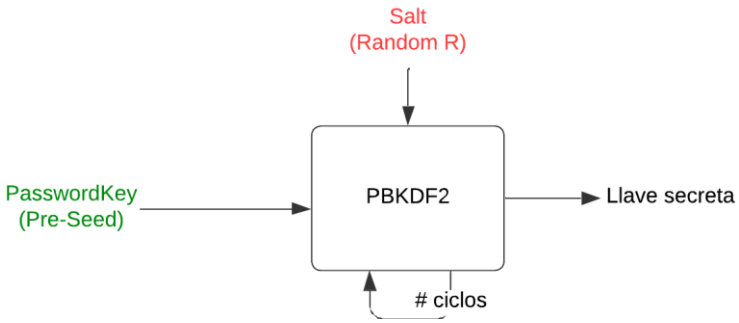


Figura 1: Generación de la llave secreta con PBKDF2

Nótese que, hasta ahora, el satélite es el único que puede generar la llave secreta, ya que inicialmente ya cuenta con los dos parámetros para generarla. Sin embargo, el cliente no puede, ya que le hace falta la Salt, que se genera con el numero random generado por el satélite. Por ello, se tiene que enviar el random generado cuando se comuniquen. Esta comunicación se va a explicar más a adelante junto a las PSK en la sección *4.4 Autenticación y compartición de llaves con PSK*.

4. Uso de PSK para asegurar el entorno

4.1 Motivación de uso de las PSK

Hasta acá, necesitamos compartir los parámetros para la base. Sin embargo, como antes mencionamos hay un problema, puesto que las compartimos en un entorno inseguro. Cualquiera que este escuchando la transmisión entre el satélite y la base, puede hacerse pasar por cualquiera de los dos, interceptando la comunicación, un ataque conocido como Man-In-The-Middle (MitM).

Para solucionar este problema, generalmente se recurre al uso clásico de certificados digitales, como en TLS 1.2 (véase " SSL, TLS, HTTPS Explained" de ByteByteGo, 2022). O también se utiliza infraestructura de clave pública (PKI) para autenticar los dispositivos. Sin embargo, el uso de certificados digitales en TLS 1.2 requiere llaves y encriptación asimétricas, que son la parte más pesada, lenta y complicada de cualquier protocolo seguro de autenticación. Por esta razón, recurrimos al uso de PSK como alternativa a los certificados digitales para autenticar ambos nodos de la comunicación, tal como se hace en TLS 1.3.

4.2 ¿Que son las PSK?

En general, vamos a autenticar los dispositivos con PSK, ¿pero que son? Las PSK, como su nombre indica, son llaves que se comparten antes de iniciar el HandShake entre el cliente y el servidor. Esto quiere decir que, el cliente y el servidor ya tienen acordada una llave que los identifica a ambos antes de enviarse información importante, al igual de cómo funciona un certificado. Todo ello, sin necesidad de intercambiar claves durante el proceso de establecimiento de la conexión.

4.3 Modos de compartición de las PSK

La compartición de las PSK antes de la comunicación se puede hacer de dos maneras: una es llamada "External PSK-Share" y la otra es llamada "Session Resumption". Para ser breves, solo vamos a explicar la que nosotros usamos, que es el método "External PSK-Share". El método "Session Resumption" se puede consultar mejor en IBM (2024), que en resumen es el método usado en conexiones con TLS 1.3.

El método "External PSK-Share", como se menciona en Housley, Hoyland, Sethi y Wood (2022), “Las PSK externas son claves secretas simétricas que se proporcionan a la implementación del protocolo TLS como entradas externas. Los PSK externos se proporcionan fuera de banda. *Traducido*”. Hay varios métodos para proporcionar las PSK fuera de banda, que se explican mejor en la sección 5 "External PSKs in Practice" del mismo documento. Sin embargo, nosotros decidimos implementar el método cargando la PSK manualmente en la memoria de ambos dispositivos, ya que es mucho más eficiente para la comunicación entre el satélite y la base, y además es más seguro que cualquier otra posible opción.

4.4 Autenticación y compartición de llaves con PSK

Hasta este punto, ya sabemos que son las PSK y como las pueden conocer el cliente y el servidor con anterioridad, pero no hemos explicado porque funcionan como mecanismo de autenticación y como nos pueden ayudar para que el cliente pueda generar su llave secreta usando PBKDF. Para una mejor compresión, usaremos la figura 1:

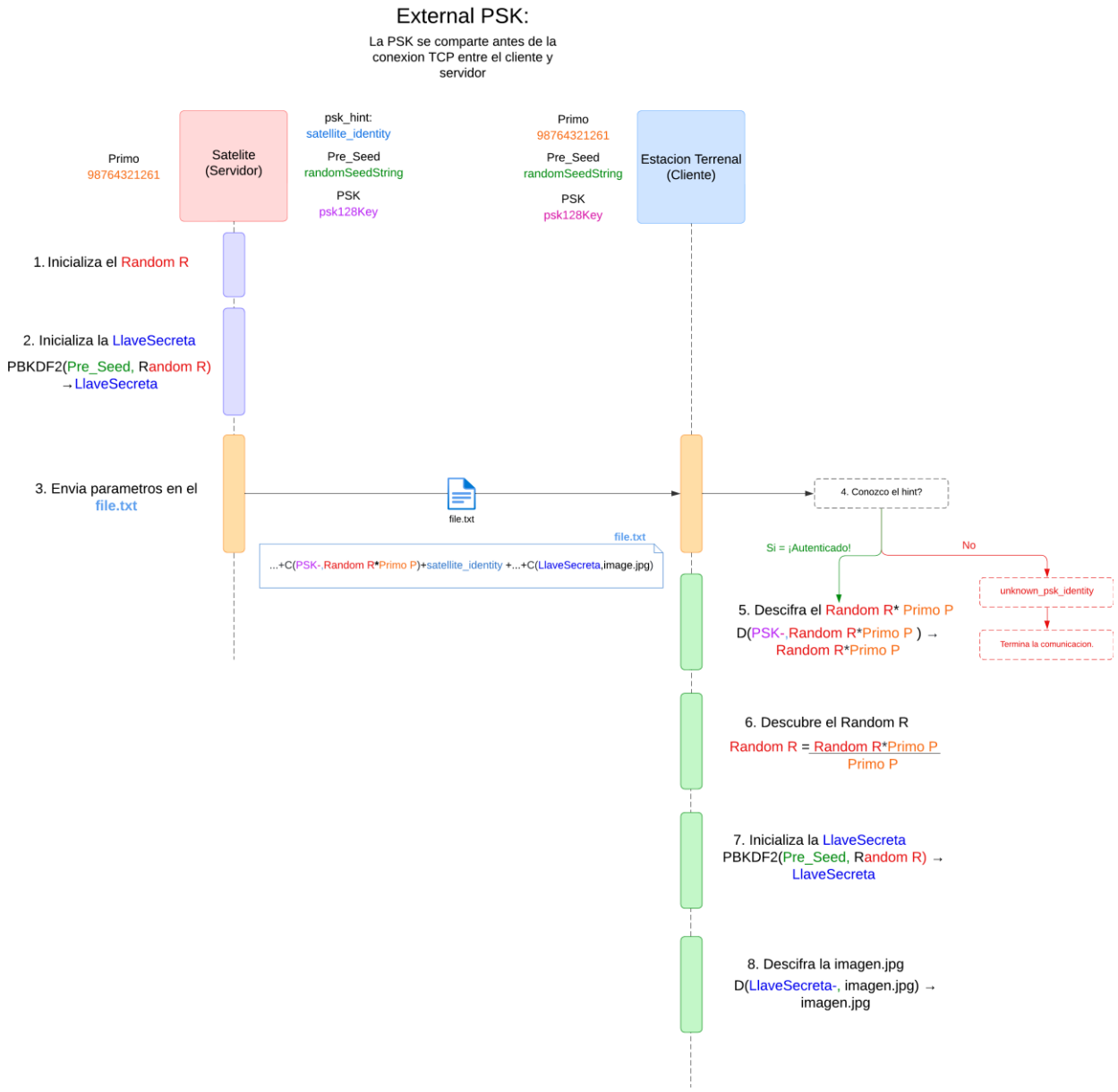


Figura 2: Proceso de autenticación y compartición de parámetros con PSK

En orden de explicar la figura 1, profundizaremos cada parte de la comunicación. Empecemos con algunos conceptos básicos.

- **psk_hint:** Funciona como una ayuda para que el cliente pueda escoger la PSK correspondiente para la comunicación con el servidor que se quiere comunicar. Al igual que las PSK, el cliente debe conocer con anterioridad los identity_hint válidos, con el fin de que pueda asociar la PSK correspondiente con el servidor que conoce.
- **C(key-,texto_plano):** Esta notación significa que se cifra con la llave key(-) privada, el texto plano. Esta notación es una abreviatura para denotar que lo que hay en el file.txt en esta parte es un texto cifrado con una llave key.

- **D(key-,texto_cifrado):** Esta notación significa que se descifra con la llave key(-) privada, el texto cifrado anteriormente. Esta notación es una abreviatura para denotar que se está descifrando alguna variable con una llave key.

Aclaración: En la práctica común del uso de PSK, los servidores tienen un conjunto de identidades conocidas de clientes, y solo aceptan las que tienen guardadas o las que ya conocen. Pasa exactamente lo mismo con los clientes, tienen un conjunto de PSK y usan la que está asociada al hint que el servidor les proporciono. Como nuestro reto solo implica la comunicación de un cliente y un solo servidor, cualquier tercero que intente pasar hints o identidades diferentes, será rechazado por la comunicación, debido a que el cliente solo conoce un único Hint y el servidor solo conoce una única identidad.

Sigamos ahora con cada parte de la comunicación:

1. **Iniciar el random:** Por cada vez que se vaya a enviar una imagen, el satelite crea un numero aleatorio diferente, asegurando que nuestra llave secreta siemopre sea diferente por cada vez que se vaya a enviar una imagen a la base terrenal.
2. **Crear la llave secreta:** Una vez creado el random, se procede a crear una llave secreta con PBKDF como se explicó en la sección 3 *Generación de llaves secretas*.
3. **Enviar file.txt:** Acá empieza la única comunicación del satélite con la base terrenal. En este archivo se mandan los parámetros para que la base genere su llave secreta y en el mismo archivo, escribe la imagen encriptada.
    a. **Factor entre el primo P y random R:** Fíjese que no mandamos el numero random en plano dentro del archivo. Por el contrario, usamos dos capas de seguridad. La primera capa es trivial, el numero va cifrado con la PSK conocida por ambos nodos. La segunda capa, es basada en la multiplicación de primos grandes de RSA. En este caso, nuestro factor es entre el numero random y el número primo. Esta estrategia nos ayuda a ofuscar el numero random, dándonos una capa extra de seguridad en dado caso que logren vulnerar nuestra PSK. Por ejemplo, si por alguna razón alguien logra descifrar el producto Primo*Random, va a tener que usar muchos recursos computacionales para factorizar ese número grande y lograr obtener el Random generado.
4. **Autenticación con el Hint:** La base verifica si el hint que se le paso en file.txt es conocido, en caso de que si, lo autentica y usa la PSK asociada con el hint conocido para descifrar el producto Primo*Random. Si no lo conoce, termina la conexión y no descifra nada, ya que se interpreta como un dispositivo no conocido para la base.
    a. **Man-In-The-Middle:** Si alguien logra interceptar nuestro file.txt, no puede saber cómo desencriptar el producto Primo*Random, ya que no tiene la PSK. Además, si intenta mandar un producto modificado para acordar la llave con la base, a partir del producto alterado, la llave que va a generar va a ser errónea, ya que el numero va a ser diferente al creado por el satélite y por lo tanto la base no podrá acordar la llave correcta. Por otro lado, si el atacante logra acordar un numero random con la base, aún tiene que conocer la semilla Pre_Seed, para

generar una llave correcta con PBKDF2, de lo contrario, aun seguirá generando llaves distintas.

5. **Descifrar Producto*Random:** En este paso, solamente se usa la PSK para descifrar el producto de estos dos números. Nuevamente, si la autenticación es correcta, se usa la PSK correspondiente al Hint proporcionado por el servidor y lo puede descifrar correctamente.
6. **Conocer el Random:** En esta parte, la base ya logra conocer su random dividiendo el producto por el numero primo guardado:

$$Random = \frac{PRimo \cdot Random}{PRimo}$$

de esta manera ya tiene la Salt para poder generar en el siguiente paso su llave secreta con PBKF.

7. **Generar llave secreta en la base:** Ya conocido el Random y la Pre_Seed, se procede a generar la llave secreta como se explicó en anteriormente en la 3 *Generación de llaves secretas.*
8. **Descifrar imagen:** Una vez generada la llave secreta, leemos el mismo archivo file.txt que contiene la imagen cifrada, para empezar a descifrarla. El proceso de descifrado se va a explicar a continuación.

5. Estrategia de cifrado

La estrategia de cifrado de la solución propuesta consiste en dividir el archivo en bloques más pequeños y cifrarlos individualmente utilizando el algoritmo AES en modo GCM. A continuación, se explica detalladamente las partes del cifrado de datos.

Se propone un entorno seguro inicializando el RNG que provee de generación segura de números aleatorios para establecer vectores de inicialización (IV) seguros para cada uno de los bloques de datos a cifrar. Luego, el archivo de entrada se divide en bloques de tamaño uniforme. Este tamaño se calcula en función de:

$$Min\left(\frac{Tamaño\ total\ del\ archivo\ a\ cifrar}{\#\ hilos\ disponibles\ para\ cifrado}, \frac{3GB}{\#\ hilos\ disponibles\ para\ cifrado}\right)$$

Asegurando que en ningún momento se cargue en memoria RAM más de 3gb de los datos a cifrar (respetando las limitaciones computacionales del sistema embebido para el que se diseñó la solución presentada). Por otro lado, si el archivo pesa menos de 3gb, se podrá hacer la división equivalente del tamaño total del archivo entre el número de threads. La optimización en términos de memoria-tiempo de ejecución mediante paralelismo con la que se cifra la información se explicará más adelante.

A continuación, se da un resumen del proceso de cifrado detallado:

1. **Inicialización:** se inicializa el generador de números aleatorios (RNG) y se prepara la clave criptográfica.
2. **División de Datos:** se calcula el tamaño de los bloques y se divide el archivo de entrada en estos bloques.
3. **Cifrado Paralelo:** cada hilo cifra un bloque de datos utilizando AES-GCM. Los IVs y tags de autenticación se generan para cada bloque.
4. **Escritura Secuencial:** los bloques cifrados se escriben en el archivo de salida en orden secuencial. Se sincroniza la escritura para asegurar el orden correcto de los bloques.

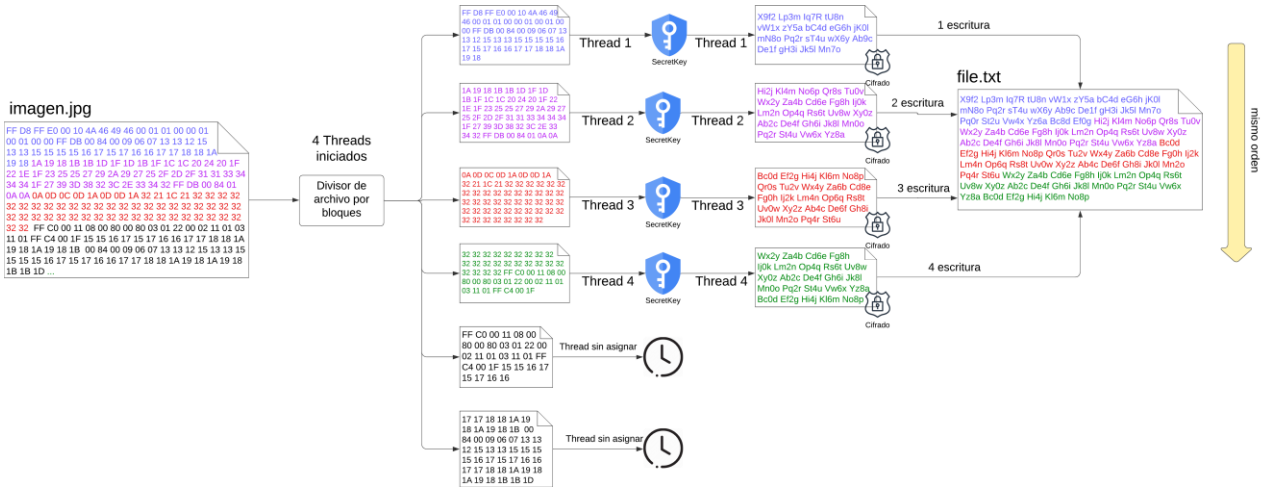


Figura 3: Diagrama de explicación de la estrategia de cifrado con threads

Nótese que la figura 3, es un diagrama meramente de ejemplo para el cifrado, en donde se usan solamente 4 threads para cifrar una imagen. Sin embargo, a partir de varias pruebas de consumo de memoria, decidimos que el uso de threads más optimo sin pasar las 4GB de RAM fueron 10 threads. Igualmente, las implicaciones de memoria y consumo de recursos de cifrar por bloques de la manera presentada en la figura 3, se explicarán más a fondo en la siguiente sección.

5.1 Optimización y uso de memoria del encriptado con threads

Cada hilo cifrará su bloque de datos usando el algoritmo AES/GCM256, con los parámetros previamente generados. Además, este modo no solo cifra los datos, también genera un tag de autenticación que se utilizará para verificar la integridad de los datos durante el descifrado. A continuación, cuando un hilo termina de cifrar, no podrá escribir en el archivo hasta que los hilos anteriores a él lo hagan, esto para garantizar que la escritura del cifrado sea correcta (ej: evitando que el bloque IV se escriba previo al bloque III). Es decir, para cifrar se usa un esquema paralelo, pero la escritura del mismo se debe hacer en ejecución secuencial.

Los beneficios de la optimización por threads planteada son:

- **Control del Tamaño del Bloque:** dividir el archivo en bloques más pequeños permite manejar eficientemente archivos grandes sin necesidad de cargar todo el archivo en memoria (requerimiento crucial debido al límite estricto de memoria del sistema embebido).

- **Memoria Temporal por Hilo:** cada hilo utiliza una cantidad fija y predecible de memoria para procesar su bloque de datos. Al asegurar que cada bloque y su procesamiento se mantengan dentro de límites específicos de memoria, se evita superar los 4 GB de RAM. Para ser más específicos, nuestro algoritmo tiene un limite de memoria de 2.6GB, en donde como máximo, cada thread de los 10, usa 280Mb como máximo para hacer el cifrado de su bloque correspondiente. Además, una vez el hilo haya escrito los datos cifrados en el archivo, se libera la memoria del mismo para que puedan ejecutarse nuevos threads. Es decir, la memoria se irá liberando dinámicamente a medida que avance la ejecución del programa sin pasar el limite impuesto anteriormente.

En general, se logra una reducción del tiempo de procesamiento debido a la paralelización, ya que se reduce el tiempo necesario para cifrar grandes volúmenes de datos al distribuir el trabajo entre múltiples hilos. Esto gracias a asignar bloques de tamaño uniforme a los hilos, se evita la sobrecarga en cualquier hilo individual, garantizando un uso eficiente de los recursos del sistema.

6. Estrategia de descifrado

La estrategia de descifrado de la solución propuesta consiste en leer los datos cifrados en bloques, descifrarlos individualmente utilizando el algoritmo AES en modo GCM y verificar la integridad de los datos mediante los tags de autenticación.

El descifrado sigue el siguiente flujo: Inicialmente, se lee el archivo cifrado en bloques del mismo tamaño que los utilizados durante el proceso de cifrado. Para cada bloque, se leen el vector de inicialización (IV) y el tag de autenticación que fueron generados durante el cifrado. Luego, se crean múltiples hilos de ejecución para manejar el descifrado de los bloques de datos. Cada hilo se encarga de descifrar un bloque específico, la estrategia del paralelismo empleado es similar al cifrado, así como todos sus beneficios. De manera similar al cifrado, aunque el descifrado se realiza en paralelo, la escritura de los bloques descifrados en el archivo de salida se realiza de manera secuencial para asegurar que los bloques se escriban en el orden correcto. Finalmente, los bloques descifrados se escriben en orden y secuencialmente en un archivo de salida, recuperando los datos originales en su forma íntegra.

En resumen, se puede observar que el mismo método de descifrado con threads, es el mismo que se hizo con el cifrado, por esta razón, puede guiarse nuevamente de la figura 3, para verlo de una manera más grafica. Lo único que cambia, es que es como si se leyera la figura al revés, empezamos con un texto cifrado, se divide por bloques, y luego cada thread descifra cada bloque.

7. Estrategia para uso de llaves dinámicas

La estrategia implementada para el uso de llaves dinámicas se basa en la generación de una nueva llave para cada sesión de comunicación. Específicamente, cada imagen transmitida entre el satélite y la base tiene asignada una llave de encriptación y desencriptación única. Este enfoque tiene múltiples ventajas, principalmente en términos de seguridad y eficiencia.

La manera en que nuestras claves se generan dinámicamente por cada vez que se envié una imagen, es gracias al número random que se multiplica con el primo por cada intento de conexión entre la base y el satélite. Este número, al ser la Salt que se usa para el método PBKDF, hace que todas las llaves generadas por este algoritmo sean diferentes a pesar de que se utilice la misma semilla. Véase la **Figura 1** para recordar los parámetros de nuestro generador de llaves

Uno de los beneficios clave de esta estrategia es la **secrecía hacia adelante** (Forward secrecy). Esto significa que incluso si una llave de encriptación se ve comprometida en algún momento, la información en las sesiones anteriores o futuras no queda expuesta. Cada llave es válida solo para una sesión específica, por lo que la violación de una llave no afecta las comunicaciones pasadas ni futuras. Esto mejora significativamente la seguridad de las comunicaciones al minimizar el impacto potencial de una brecha de seguridad.

8. Las correcciones más significativas

Con el fin de cumplir con todos los requerimientos, se hicieron los siugientes cambios que fueron más significativos en nuestro algoritmo:

1. **Conexión única del satélite a la base:** Debido a las restricciones de comunicación, logramos reducir el 3-way-Handshake propuesto anteriormente a solo una conexión. Como se pudo observar, la única comunicación que hacemos es el envío de un solo archivo que contiene los parámetros para que la base pueda generar la llave secreta, sumado a la imagen encriptada.
2. **Generación de llaves secretas:** Como consecuencia directa al cambio anterior, tuvimos que dejar de lado el uso de ECDH(Eliptic-Curve-Diffie-Hellman) para compartir las llaves, ya que este método suponía comunicación bidireccional para obtener ambas llaves. En lugar de usar ECDH, usamos un algoritmo basado en contraseñas llamado PBKD2, que nos permite crear llaves secretas a partir de una salt y una semilla dada. Este algoritmo, nos permitía acordar las llaves usando una sola conexión entre el satélite y la base como se ha explicado en el documento, además de que nos permitía crear llaves dinámicamente.

9. Librerías utilizadas

- **WolfSSL:** Hemos elegido la biblioteca wolfSSL para nuestro proyecto de encriptación en C++ que utiliza PSK, AES256 y GCM debido a su sólida seguridad y cumplimiento de estándares modernos, su rendimiento eficiente en sistemas con recursos limitados, y su amplia compatibilidad con diversas plataformas. La biblioteca ofrece una implementación optimizada de algoritmos criptográficos robustos y proporciona una

API bien documentada que facilita la integración y el desarrollo. Además, tiene varios módulos de implementación para dispositivos IoT y funciones de optimización de recursos para kernel de linux y CPU's ARM, convirtiéndose en una herramienta muy versátil que podemos seguir usando en siguientes retos.

- **GTest:** Esta librería se usó para hacer test-unitarios de las funciones críticas del programa, y de esta manera hacer mejor cobertura.

10. Estrategia de verificación y validación usada para medir la calidad interna del código

10.1 Pruebas de rendimiento: por threads

10.1.1 100MB

Tipo Archivo	**Tamaño** (MB)	**Threads**	**RAM consumida encriptar**	**RAM consumida desencriptar**	**Total RAM**	**Tiempo encriptación** (s)	**Tiempo Desencriptando** (s)
.tif	133,30	5,00	59904,00	56704,00	116608,00	1,29	1,28
.tif	133,30	10,00	33704,00	30592,00	64296,00	1,33	1,43
.tif	133,30	20,00	20864,00	23828,00	44692,00	1,40	1,44
.tif	133,30	50,00	13060,00	12116,00	25176,00	1,46	1,22

10.1.2 500MB

Tipo Archivo	**Tamaño** (MB)	**Threads**	**RAM consumida encriptar**	**RAM consumida desencriptar**	**Total RAM**
.tif	696,80	5,00	280064,00	276736,00	556800
.tif	696,80	10,00	143872,00	140736,00	284608
.tif	696,80	20,00	75904,00	72576,00	148480
.tif	696,80	50,00	34944,00	45376,00	80320

10.1.3 1GB

Tipo Archivo	**Tamaño** (MB)	**Threads**	**RAM consumida** encriptar	**RAM consumida desencriptar**	**Total RAM**
.jpg	1100,00	5,00	423040,00	419584,00	842624
.jpg	1100,00	10,00	215472,00	212204,00	427676
.jpg	1100,00	20,00	111616,00	108516,00	220132
.jpg	1100,00	50,00	49340,00	45952,00	95292

10.1.4 5GB

Tipo Archivo	**Tamaño** (MB)	**Threads**	**RAM consumida** encriptar	**RAM consumida desencriptar**	**Total RAM**

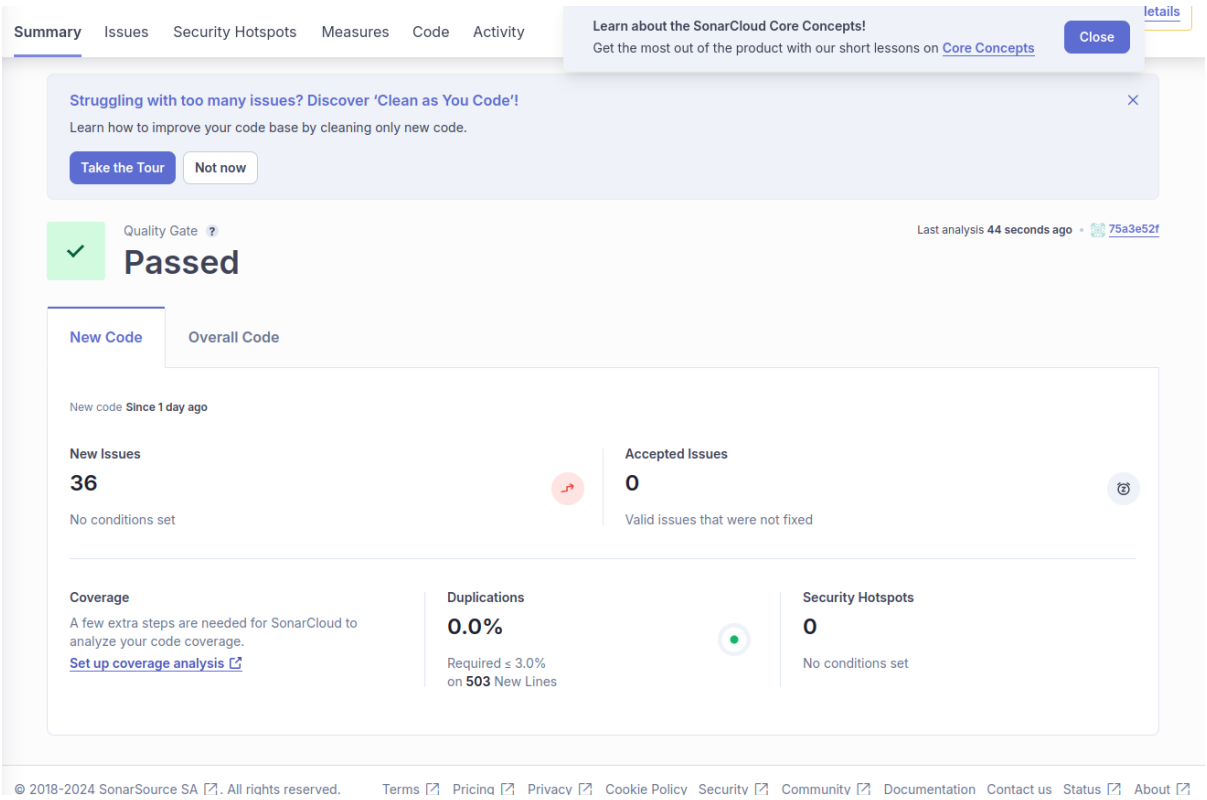
.jpg	5300	5	2083968,00	2080512,00	4164480,00
.jpg	5300	10	1045760,00	1042560,00	2088320,00
.jpg	5300	20	526848,00	526848,00	1053696,00
.jpg	5300	50	215456,00	212260,00	427716,00

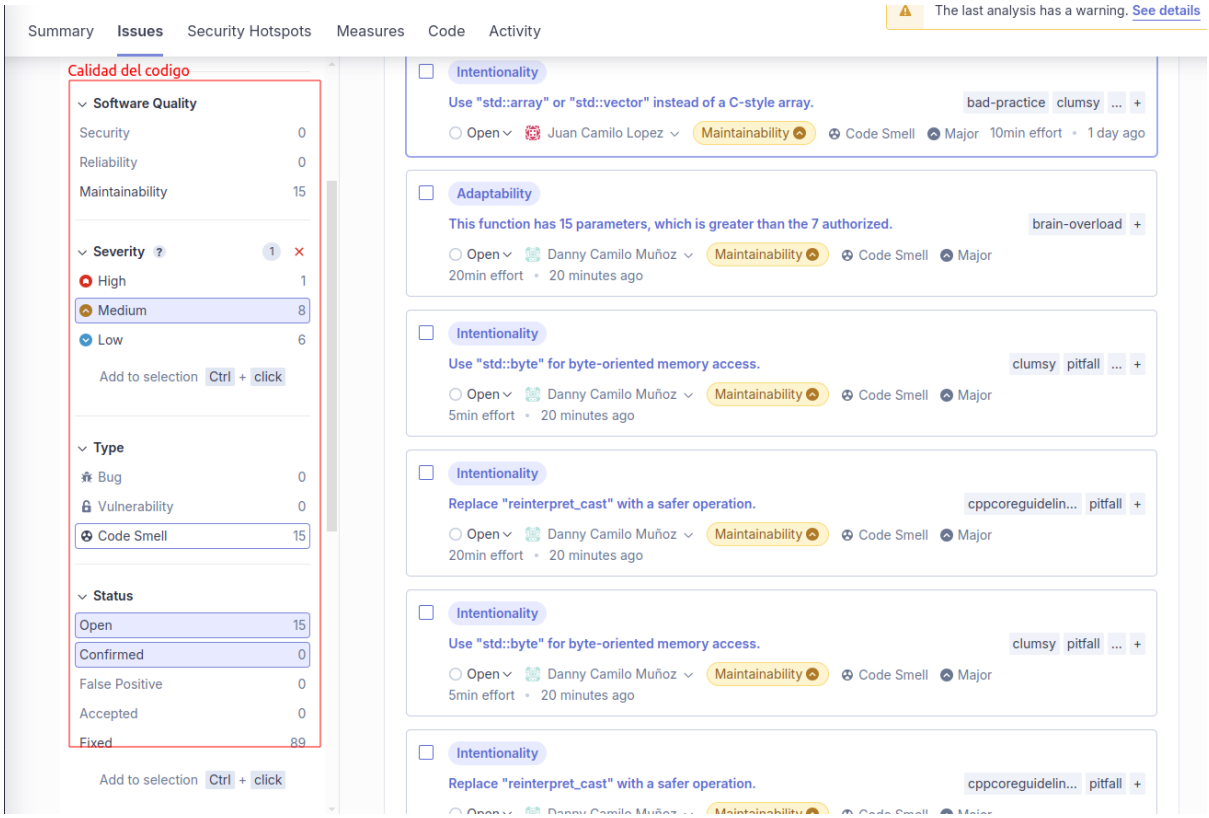
10.1.5 10GB

Tipo Archivo	Tamaño	Threads	RAM consumida	Tiempo encriptación	Tiempo Desencriptando
		5			
		10			
		20			
		30			

10.2 Verificación calidad interna del código

Como estrategia de validación, usamos el analizador estatico SonarCloud que nos muestra las siguientes métricas:





Igualmente, usamos la librería de test unitarios de código GTest de Google para verificar algunas de las funciones críticas de nuestra solución. Para revisarlas, asegúrese de revisar el readme.md del repositorio con las indicaciones propias para ver las pruebas realizadas.

11. Referencias

Blue Goat Cyber. (2022). *Elliptic Curve Diffie-Hellman (ECDH)*. Recuperado de https://bluegoatcyber.com/blog/elliptic-curve-diffie-hellman-ecdh/.

ByteByteGo. (2022, December 8). *SSL, TLS, HTTPS Explained* [Video]. YouTube. https://www.youtube.com/watch?v=j9QmMEWmcfo

IBM. (2024). *Session Resumption with Pre-shared Key*. Recuperado de https://www.ibm.com/docs/en/sdk-java-technology/8?topic=handshake-session-resumption-pre-shared-key.

Housley, R., Hoyland, J., Sethi, M., & Wood, C. A. (2022). *External PSK Usage Guidance*. RFC 9257. Internet Engineering Task Force (IETF). Recuperado de https://datatracker.ietf.org/doc/rfc9257/.

Eronen, P., & Tschofenig, H. (2005). *Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)*. RFC 4279. Network Working Group. Recuperado de https://www.rfc-editor.org/rfc/rfc4279.