UNIVERSIDAD DE MENDOZA

FACULTAD DE INGENIERÍA

TESIS DE

MAESTRÍA EN TELEINFORMÁTICA

Criptografía Post-Cuántica Aplicada a Entornos de Producción Open Source

Autor: Ing. Córdoba Diego

Director: Dr. Méndez Garabetti Miguel

CoDirector: Mg. Ing. García Guibout, Jorge S.

Mendoza - Junio de 2022

CRIPTOGRAFÍA POST-CUÁNTICA

APLICADA EN ENTORNOS DE PRODUCCIÓN OPEN SOURCE *(v5.0)*

Ing. Diego Córdoba

Libro fue escrito en LaTeX, un sistema de preparación de documentos libre.

Agradecimientos

Quiero agradecer especialmente a Andrea, mi compañera de vida, por su ayuda durante las correcciones de este trabajo, por su continuo apoyo en todo lo que hago, y por su amor incondicional.

A mis padres, por estar siempre presentes en todos los hitos de mi vida, animándome a seguir adelante con cada locura que quise emprender; por enseñarme que el trabajo y la perseverancia son la base del éxito, y por demostrarme que la humildad es lo que hace grande a las personas.

A mis hermanos, por haber estado ahí siempre que los necesité, y por formar parte de lo que soy.

A Miguel y Jorge por el tiempo y la inestimable orientación brindados durante la elaboración de esta tesis.

A mis colegas y alumnos de la Facultado de Ingeniería de la Universidad de Mendoza por su compañerismo y amistad, y por ayudarme a mejorar como profesional y como persona.

A mis alumnos y amigos de JuncoTIC, los que están a la vuelta de la esquina y aquellos que están del otro lado del mundo, por haberme alentado y esperado el tiempo que me llevó terminar esta tesis.

Finalmente, y no menos importante, a las comunidades de software libre y de código abierto por las amistades que me dieron durante tantos años de colaboración y aprendizaje, y sobre todo por liberar código y conocimientos desinteresadamente, sin los cuales este proyecto nunca hubiera existido.

Índice general

RESUMEN

Actualmente la mayoría de los protocolos de aplicación en Internet delegan su seguridad a la capa TLS por medio de implementaciones de software tales como OpenSSL y wolfSSL. Esta capa genera un canal seguro entre cliente y servidor para ofrecer privacidad, autenticidad e integridad a los datos que transitan la red. TLS utiliza criptografía asimétrica para lograr la autenticación e intercambio de las claves. Estos métodos se basan en supuestos de complejidad computacional, por lo que hacen uso de funciones matemáticas muy simples de calcular, pero extremadamente difíciles de revertir haciendo uso de los procesadores actuales. No obstante, estos mecanismos se consideran vulnerables al criptoanálisis realizado desde computadoras cuánticas. Aún así, existen algoritmos asimétricos que se consideran resistentes a este tipo de ataque: los algoritmos post-cuánticos.

Este trabajo estudió el estado del arte en el campo de la criptografía post-cuántica, mencionó los avance en el proceso de estandarización y los algoritmos candidatos más prometedores en la actualidad. Además, analizó las implementaciones de software y librerías de código abierto que han alcanzado mayor nivel de madurez, y que a su vez cuentan con un desarrollo activo. Se detallaron dos implementaciones de TLS: wolfSSL y las variantes de OpenSSL provistas por el proyecto Open Quantum Safe, OQS-OpenSSL; y se realizaron experimentos de integración de estas implementaciones con HTTP y OpenVPN como protocolos de capa de Aplicación.

A pesar de los avances que este campo ha tenido en los últimos años, esta tesis da cuenta del nivel experimental de las implementaciones de software disponibles y de cómo, tanto éstas como el desarrollo de nuevos estándares, se encuentran aún en etapas tempranas de su evolución.

I INTRODUCCIÓN

I.1 PLANTEAMIENTO DEL PROBLEMA

Las comunicaciones en Internet se han vuelto indispensables tanto para la vida de las personas como para el funcionamiento de las sociedades y organizaciones. Su uso se ha incrementado sustancialmente la última década debido a las posibilidades que brindan los avances tecnológicos en computadoras, teléfonos inteligentes, y más aún con el surgimiento de paradigmas como el Internet de las cosas o la computación de nube. Esta gran interconexión habilita a usuarios, empresas y gobiernos a hacer uso de la red de redes para acceder y brindar servicios en línea a través de aplicaciones móviles, sistemas web, entre otros (Max Roser y Ortiz-Ospina, 2015). Esta tendencia se vio incrementada desde el 2020 por la pandemia del coronavirus COVID-19 (Clement, 2020).

Consecuentemente la seguridad en las transacciones en Internet está tomando especial relevancia, particularmente en operaciones comerciales, el acceso a la banca electrónica, el uso de billeteras virtuales, el uso de sistemas de mensajería y correo electrónico, y servicios que requieran de autenticación. La seguridad en estas comunicaciones permite garantizar la confidencialidad, integridad y autenticidad de los datos del usuario en su viaje entre los nodos que conforman el enlace. Estos requisitos se logran generalmente mediante el uso de mecanismos criptográficos tales como las herramientas de cifrado, funciones de hashing o resumen, y firmas digitales (Stallings, 2005).

La suite de protocolos TCP/IP en la que está basada Internet (Comer, 2014) divide la funcionalidad de las comunicaciones en capas. TCP/IP provee las especificaciones de la comunicación de datos de extremo a extremo, es-

tablece cómo deben empaquetarse, direccionarse, transmitirse, enrutarse y recibirse los datos. Las capas del modelo TCP/IP, también conocida como Arquitectura TCP/IP o Arquitectura de Internet, son cuatro (Peterson y Davie, 2007), a saber:

- Aplicación: administra el intercambio de datos entre procesos.
- Transporte: administra la comunicación entre los equipos extremos.
- Red: provee la comunicación entre redes independientes.
- Enlace: brinda los mecanismos para la transmisión y recepción de datos en un segmento simple de red.

Cada capa opera según reglas definidas por protocolos, y las computadoras en la red hacen uso de implementaciones de software de estos protocolos para gestionar las comunicaciones. Los protocolos de Capa de Aplicación más utilizados no implementan mecanismos de seguridad en sí mismos, sino que delegan estas funcionalidades a una subcapa denominada Capa de Conexión Segura (SSL, por sus siglas en inglés) (Freier y Karlton, 2011) o Seguridad en Capa de Transporte (TLS, por sus siglas en inglés) (Dierks y Rescorla, 2008). Así, por ejemplo, un protocolo común como HTTP (Fielding et al., 2014) combinado con la capa de seguridad SSL/TLS conforma el protocolo seguro HTTPS (HTTP over/sobre SSL/TLS) (Rescorla, 2000). Lo mismo ocurre con otros protocolos de Aplicación de uso común como SMTP (Klensin, 2001), IMAP (Crispin, 2003), OpenVPN (OpenVPN Inc., 2018) o FTP (Postel y Reynolds, 1985) .

Debido al incremento de adopción del protocolo HTTPS como alternativa a HTTP, buscadores como Google comenzaron a tener en cuenta el su uso en los algoritmos de posicionamiento (Bahajji e Illyes, 2014), y navegadores como Chrome o Firefox comenzaron a advertir a los usuarios cuando los sitios accedidos no fueran seguros (Cimpanu, 2017; Chromium Blog, 2018). Incluso Firefox permite bloquear el acceso a sitios web que no provean se-

guridad en HTTP mediante el modo “HTTPS-only” (Sólo-HTTPS) disponible desde la versión 83 (Kerschbaumer et al., 2021).

SSL/TLS le brinda a HTTP algoritmos de autenticación basados en certificados digitales X.509 (ITU-T, 2019) y criptografía asimétrica (Lucena López, 2011). Este tipo de algoritmos se basan generalmente en funciones matemáticas que hacen uso de supuestos de complejidad computacional, es decir, problemas matemáticos muy simples de calcular, pero extremadamente difíciles de revertir. Así la generación de claves asimétricas, por ejemplo, puede llevarse a cabo multiplicando dos números primos grandes. El producto de estos dos valores es relativamente sencillo de computar para un procesador actual, pero encontrar estos dos números primos que dieron origen al producto resulta una tarea sumamente difícil aplicando algoritmos de factorización y de fuerza bruta que hagan uso de las capacidades de cálculo disponibles hoy en día (Bernstein y Lange, 2017). Por otro lado, una computadora cuántica, gracias a las características intrínsecas de su arquitectura, podría encontrar estas claves de cifrado de manera más eficiente que un procesador clásico, vulnerando así la seguridad brindada por los algoritmos asimétricos tradicionales (Bernstein et al., 2009).

Una computadora cuántica podría encontrar claves de cifrado utilizando algoritmos cuánticos especialmente diseñados para este tipo de procesadores. Uno de los algoritmos cuánticos con mayor relevancia en el campo de la criptografía es el algoritmo de Shor (Shor, 1997). Se lo considera el primer algoritmo cuántico no trivial (es decir, que plantea una solución compleja y no obvia) que demostró un potencial de crecimiento exponencial de velocidad sobre los mecanismos clásicos (Chu, 2016). Este algoritmo permite obtener el resultado de manera estocástica y con un determinado grado de acierto según la cantidad de iteraciones a la que se lo someta.

El algoritmo de Shor logra encontrar una clave RSA (Rivest et al., 1987) descomponiéndola en factores en un tiempo $O((\log N)^3)$, siendo N el numero primo que representa la clave pública. Por su parte, los algoritmos clásicos no pueden realizar dicha factorización en un tiempo menor a $O((\log N)k)$ para ningún k, por lo que RSA sigue siendo considerado, en la actualidad, un algoritmo seguro (Bernstein et al., 2009; Weisstein, 2005). Isaac Chuang en diciembre de 2001, liderando un grupo de trabajo de computación cuántica en IBM, logró factorizar el número 15 mediante una computadora cuántica de 7 qubits (Vandersypen et al., 2001), y en marzo de 2016 un grupo de investigadores del MIT, entre los que se encontraba el mismo Chuang, pudo factorizar el numero 15 con un 99 % de certeza en una computadora cuántica de 5 qubits, cada uno representado por un átomo individual (Chu, 2016). De aquí se deduce que la aparición de las primeras computadoras cuánticas de cientos de qubits con una reducción aceptable del ruido cuántico dejarían obsoletos a la mayoría de los algoritmos asimétricos actuales (Bernstein et al., 2009; ETSI, 2015; Takagi, 2016; Perlner y Cooper, 2009).

Para resolver este potencial problema se han diseñado algoritmos asimétricos resistentes al criptoanálisis cuántico, conocidos como algoritmos post-cuánticos (Bernstein y Lange, 2017). Estos algoritmos pueden ejecutarse en computadoras tradicionales y ser desplegados en las redes de datos actuales, facilitando de esta forma su despliegue. Esto les da una gran ventaja sobre otros mecanismos de intercambio de claves tales como la criptografía cuántica, que requieren montar una nueva infraestructura de comunicaciones (Gisin et al., 2002).

Existen varias implementaciones de software de SSL/TLS, sin embargo ninguna soporta algoritmos post-cuánticos de manera nativa. Las implementaciones de código abierto más utilizadas son OpenSSL (OpenSSL Software Foundation, 2018) y wolfSSL (wolfSSL, 2018), una alternativa centrada en la eficiencia para su ejecución en sistemas embebidos y dispositivos IoT.

Aunque OpenSSL no incluye algoritmos post-cuánticos de manera nativa, el proyecto Open Quantum Safe (Stebila y Mosca, 2017) ha generado un *fork* que soporta una gran variedad de algoritmos asimétricos y de intercambio de claves post-cuánticos integrando OpenSSL mediante una biblioteca de código denominada *liboqs* (OQS, 2021a). Esta librería ha sido usada por proyectos de terceros como la VPN experimental de Microsoft, PQCrypto-VPN (Microsoft, 2018), o el cliente beta de VPN de Mullvad (Amagicom AB, 2017), que utiliza intercambio de claves post-cuántico. Por otro lado, wolfSSL recientemente incorporó en su implementación a NTRU, un algoritmo asimétrico post-cuántico que se cree es resistente a ataques cuánticos, y provee una seguridad similar a la de RSA pero haciendo uso de claves más cortas y mejor rendimiento (Hoffstein et al., 1998).

En esta tesis se estudiaron wolfSSL y dos versiones de OQS-OpenSSL. En todos los casos se realizaron experimentos que permitieron verificar el uso de cipher suites (conjunto de mecanismos criptográficos que dan seguridad a la conexión) y algoritmos de intercambio de claves resistentes al criptoanálisis cuántico durante negociación TLS. En cuanto a OQS-OpenSSL se analizó una integración con dos implementaciones de servicio HTTP, Apache (The Apache Software Foundation, 2018) y NGINX (Nginx, Inc., 2018), y luego se montó una prueba de concepto de un enlace VPN utilizando PQCrypto-VPN. Respecto a wolfSSL se llevó a cabo una adaptación básica de un servidor Apache para lograr su integración con esta implementación, y se logró la negociación del algoritmo NTRU (N-Th Degree TRUncated Polynomial Ring, Anillo Polinomial Truncado de N-ésimo grado) (Baktu, 2017) en el establecimiento de una conexión HTTPS. Además, se logró montar una prueba de concepto de la integración de wolfSSL con OpenVPN aunque, en este último caso, sin la incorporación de criptografía post-cuántica.

I.2 OBJETIVOS

El presente trabajo de investigación se centra en el análisis de algunas implementaciones de SSL/TLS que incorporan algoritmos criptográficos post-cuánticos en la negociación cliente-servidor, y que a su vez se encuentran en un estado relativamente activo de desarrollo.

Como objetivo general el trabajo plantea implementar pruebas de concepto de algoritmos criptográficos post-cuánticos en protocolos HTTP y VPN.

Los objetivos específicos que se desprenden de este son los siguientes.

- Realizar un análisis minucioso de las implementaciones de SSL/TLS post cuánticas que se encuentran en un estado de desarrollo más avanzado y activo, a saber, OQS-OpenSSL y wolfSSL, con la intención de determinar su nivel de estabilidad, bugs y vulnerabilidades, respecto de la última release de OpenSSL.
- Montar las pruebas de concepto de servicio HTTPS con cifrado post-cuántico utilizando Apache y/o Nginx integrado con las implementaciones OpenSSL antes mencionadas.
- Montar prueba de concepto de servicio de VPN en capa 7 basado en OpenVPN con cifrado post-cuántico mediante la combinación con las suites OpenSSL antes mencionadas.

I.3 ORGANIZACIÓN DEL TRABAJO

La presente tesis se divide en tres partes: Marco teórico, Desarrollo y Conclusiones. Se incluye un capítulo inicial, Introducción, que detalla el planteamiento del problema que aborda la tesis, especifica el objetivo general y los específicos, y describe la organización del trabajo.

La primera parte, MARCO TEÓRICO, aborda la teoría necesaria para comprender el desarrollo del trabajo. Esta parte se divide en 4 capítulos:

- El Capítulo II “CRIPTOGRAFÍA APLICADA” introduce los conceptos básicos del uso de la criptografía en redes TCP/IP, y especialmente los detalles del protocolo TLS.
- El Capítulo III “SSL/TLS” realiza una introducción a SSL/TLS y a la cipher suite QSH (Quantum Safe Hybrid).
- El Capítulo IV “COMPUTACIÓN CUÁNTICA Y SEGURIDAD INFORMÁTICA” trata sobre la amenaza que representa la computación cuántica a los algoritmos asimétricos actuales.
- El Capítulo V “CRIPTOGRAFÍA POST-CUÁNTICA” presenta los detalles de la criptografía post-cuántica utilizada en el desarrollo.
- El capítulo VI “IMPLEMENTACIONES” introduce el proyecto Open Quantum Safe, la librería liboqs, algoritmos soportados, y finalmente reseña el proyecto wolfSSL.

La segunda parte, DESARROLLO, detalla los análisis y experimentos prácticos realizados. Esta parte se divide en 4 capítulos:

- El capítulo VII “ANÁLISIS DE OQS-OPENSSL” realiza un análisis de las características de OQS-OpenSSL en sus versiones 1.0.2 y 1.1.1.
- El capítulo VIII “EXPERIMENTACIÓN: OQS” detalla los experimentos realizados con OQS-OpenSSL v1.0.2 y v1.1.1, y su integración con Apache, Nginx y OpenVPN.
- El capítulo IX “EXPERIMENTACIÓN: WOLFSSL” realiza el análisis y experimentación de wolfSSL, NTRU, Apache y OpenVPN.
- El capítulo X “OTRAS IMPLEMENTACIONES” reseña otras implementaciones de software de interés no incluidas en el presente trabajo, y reseña los trabajos relacionados con este proyecto.

Finalmente, la tercera parte, CONCLUSIONES, expone las conclusiones obtenidas del desarrollo, y algunas líneas de investigación que podrían desprenderse de esta tesis.

MARCO TEÓRICO

Esta sección presentan los conceptos fundamentales sobre los que se basa esta tesis. Se realiza una introducción a la Criptografía Aplicada y su relación con la seguridad informática. Se introducen tópicos básicos de computación cuántica y su relación con los algoritmos criptográficos. Se introducen los fundamentos de criptografía post-cuántica, los algoritmos candidatos y su proceso de estandarización. Finalmente se presentan las dos implementaciones de software que se analizarán en adelante: OQS-OpenSSL y wolfSSL.

II CRIPTOGRAFÍA APLICADA

El término **criptografía** proviene del griego *kryptós*, que significa *oculto o secreto*, y *graphé*, que significa *grafo o escritura*: *escritura oculta o secreta*. Es decir, es el estudio de las técnicas destinadas a asegurar las comunicaciones en presencia de terceros denominados adversarios o atacantes. La criptografía permite diseñar y analizar mecanismos que brinden la comunicación secreta a dos partes interesadas, utilizando un canal público considerado hostil.

Pueden distinguirse dos eras en la criptografía: la clásica y la moderna. La clásica hace referencia a las técnicas de transposición y sustitución de caracteres para ocultar mensajes. La criptografía moderna es la que surge con la era de la computación, y hace uso de conocimientos de álgebra, teoría de números, estadística, combinatoria, y hasta física cuántica para lograr su objetivo. En el presente trabajo resulta de interés ésta última.

Se define al criptoanálisis como la disciplina que se encarga de estudiar de los sistemas criptográficos con la intención de encontrar vulnerabilidades sin conocer ninguna información secreta. Ambos, criptografía y criptoanálisis, forman parte de una ciencia llamada criptología.

La seguridad de los algoritmos criptográficos modernos se basa en supuestos de complejidad computacional no demostrados, tales como operaciones matemáticas unidireccionales. De esta forma las claves secretas necesarias para el cifrado se generan mediante operaciones matemáticas muy simples de calcular pero extremadamente difíciles de revertir. Por ejemplo, una clave RSA se genera a partir de la multiplicación de dos números primos grandes. Esta operación es muy simple de realizar con cualquier computadora clásica. No obstante, teniendo el valor de este producto, un

supuesto atacante debería invertir mucho tiempo de cómputo para encontrar los dos números primos que le dieron origen. Este tiempo resulta una limitante para el atacante.

Por otro lado, los avances tecnológicos han permitido un incremento en la potencia de cálculo con técnicas como el aumento de la frecuencia de procesamiento o el uso de paralelismo. Esto le brinda a la computadora maneras de ejecutar más operaciones simples por segundo, reduciendo el tiempo de ejecución que requiere un algoritmo. El incremento de potencia beneficia a aquellas aplicaciones que necesitan realizar cálculos complejos, como es el caso del criptoanálisis. En la medida en que los procesadores se hacen más rápidos, las tareas de criptoanálisis se vuelven más eficientes, y permiten romper algoritmos criptográficos simples o cuyas claves sean relativamente cortas. Por este motivo, el incremento de la potencia de procesamiento acompaña al desarrollo e implementación de algoritmos criptográficos más complejos y/o al incremento del tamaño de las claves de cifrado sin que esto ocasione en una pérdida de rendimiento.

II.1 CRIPTOGRAFÍA SIMÉTRICA Y ASIMÉTRICA

Para entender los conceptos subyacentes a la criptografía simétrica y asimétrica primero es necesario introducir las nociones de criptosistema. Un criptosistema es un conjunto de algoritmos necesarios para implementar servicios de seguridad tales como la confidencialidad o privacidad, la autenticidad y la integridad. El esquema general de un criptosistema puede apreciarse en la Fig. 1.

Los componentes principales del mismo son:

- Emisor: entidad encargada de emitir un mensaje en texto plano.
- Sistema de cifrado: algoritmo criptográfico que generará un mensaje cifrado a partir del mensaje original y una clave.

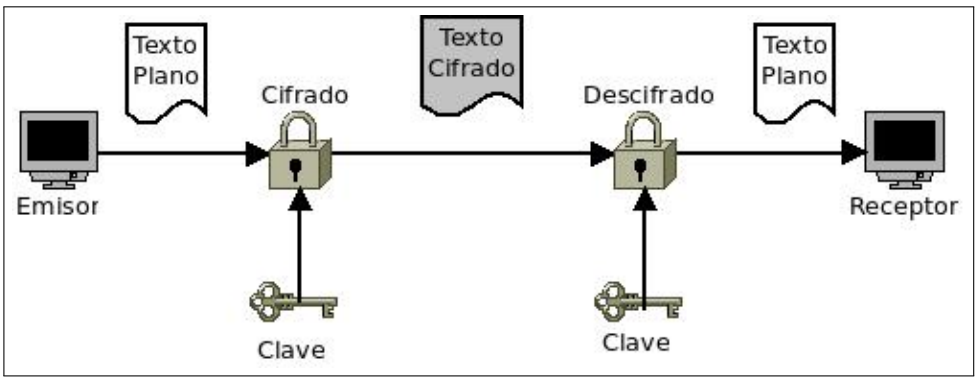

Figura 1: Criptosistema general.
Fuente: (Ramió Aguirre, 2018)

- Sistema de descifrado: algoritmo criptográfico que generará un mensaje en texto plano a partir del mensaje cifrado y una clave.
- Receptor: entidad destinataria del mensaje original.
- Mensaje: datos que el emisor envía al receptor. Según la etapa del criptosistema en la que se encuentre, puede tratarse de un mensaje en texto plano o un mensaje cifrado.
- Clave: clave o claves utilizadas para cifrar y/o descifrar mensajes.

Ahora bien, en un criptosistema puede que el algoritmo de cifrado y el de descifrado sean el mismo o no, y a su vez puede ocurrir también que las claves de cifrado y de descifrado coincidan o sean diferentes.

Cuando la claves de cifrado y de descifrado es la misma se dice que el criptosistema es simétrico o de clave secreta, y cuando la clave usada para cifrar el mensaje difiere de la que se usa para descifrarlo, se dice que el criptosistema es asimétrico o de clave pública.

II.1.1 Criptografía simétrica

Los algoritmos criptográficos simétricos a su vez pueden ser de flujo o de bloques, dependiendo de si el procedimiento de cifrado y de descifrado se lleva a cabo un caracter a la vez o en bloques de información de tamaño fijo. Un cifrador que genere un texto cifrado procesando muy pequeñas por-

ciones del texto original, por ejemplo, un caracter a la vez, es un cifrador de flujo o *stream*. Un algoritmo simétrico de flujo común es RC4. Por su parte, si el cifrado y descifrado se lleva a cabo por bloques de tamaño fijo, por ejemplo 128 bits, se trata de un cifrador simétrico de bloques. Uno de los algoritmos más utilizados en la actualidad en esta categoría es AES.

Los algoritmos simétricos de bloques a su vez pueden funcionar utilizando diferentes esquemas de encadenamiento de bloques para lograr el cifrado del conjunto completo de datos. Esos esquemas se denominan *modos de operación*. Si bien no se realizará una explicación detallada de cada modo, es importante mencionarlos para comprender la conformación de las cipher suites de SSL/TLS más adelante. Los modos más comunes utilizados son:

- ECB: Electronic Code Book
- CBC: Cipher Block Chaining
- PCBC: Propagating CBC
- CFB: Cihper FeedBack
- OFB: Output FeedBack
- CTR: Counter
- GCM: Galois Counter Mode

Uno de los principales inconvenientes de los criptosistemas simétricos es el intercambio de la clave secreta entre las partes involucradas sin que ningún adversario pueda interceptarla. Un protocolo para intercambio de claves muy utilizado en la actualidad es Diffie-Hellman, mecanismo que sentó las bases de la criptografía asimétrica (véase la Sección II.2.3).

II.1.2 Criptografía asimétrica

Los algoritmos asimétricos hacen uso de dos claves: una pública y una privada. Esto tiene una ventaja sobre los simétricos: sólo requiere que las partes involucradas intercambien sus claves públicas. Las claves públicas

suelen distribuirse en un formato encapsulado denominado certificado digital. El estándar más utilizado en este caso es el x509 (ITU-T, 2019). Este certificado incorpora, además de una clave pública, datos del propietario de dicha clave, información de validez temporal y firma digital emitida por alguna autoridad certificante. El conjunto de claves privadas, certificados digitales x509 y entidades de firma de dichos certificados da origen a lo que se conoce como Infraestructura de Clave Pública, o PKI (Public Key Infrastructure, Infraestructura de Clave Pública). Una desventaja de los algoritmos asimétricos respecto de los simétricos radica en los cálculos que realizan: las tareas de cifrado y descifrado asimétricas requieren más operaciones matemáticas, por lo que pueden perjudicar el rendimiento de los sistemas donde se implementen.

II.2 CRIPTOGRAFÍA Y SEGURIDAD INFORMÁTICA

La criptografía representa la base de las técnicas de seguridad informática en Internet. Tareas como la navegación en sitios web, la autenticación de usuario en el home banking, o en una API que accede a servicios en los que prima la privacidad, requieren de mecanismos criptográficos que garanticen la seguridad en la comunicación. Éstos requisitos deben proveerse en un amplio abanico de protocolos de red tales como HTTP, protocolos de transferencia de archivos, o de comunicaciones en túnel como el caso de las VPN (Virtual Private Network, Redes Privadas Virtuales).

Ahora bien: ¿cómo se logra la seguridad en este contexto? Para que una comunicación en Internet sea segura deben verificarse tres requisitos: privacidad, autenticidad, e integridad de los mensajes intercambiados. El receptor de los mensajes es quien verifica estos requisitos, y puede hacerlo utilizando criptografía simétrica, asimétrica, o una combinación de ambas.

II.2.1 Privacidad o confidencialidad

Este punto se refiere a implementar mecanismos criptográficos que cifren los mensajes enviados de tal modo que únicamente pueda descifrarlos el destinatario de los mismos. Así, si existiera un atacante en el canal, éste no podrá ser capaz interpretar el contenido de los mismos. La privacidad puede lograrse mediante criptografía simétrica o asimétrica.

En el caso de usar criptografía simétrica ambos nodos deberán compartir previamente una clave secreta. Esta clave permitirá a un extremo cifrar un mensaje haciendo uso de un algoritmo de cifrado simétrico, y el otro extremo podrá descifrarlo utilizando la misma clave y el algoritmo correspondiente. Un atacante no podrá acceder al contenido ya que no poseerá la clave secreta compartida por emisor y receptor. Es importante evitar que la clave secreta se vea comprometida en el medio.

Usando criptografía asimétrica ambos extremos deberán tener un juego de claves pública y privada. Un mensaje cifrado con una de las claves podrá ser descifrado únicamente con la otra clave del par. Así, el emisor podrá encriptar un mensaje usando la clave pública del receptor, y éste podrá desencriptarlo utilizando su clave privada, par de la pública usada por el emisor. Un atacante no podría descifrar el mensaje ya que no tiene acceso a la clave privada del receptor.

II.2.2 Autenticación, Integridad y No Repudio

La autenticidad y la integridad pueden verificarse usando los mismos mecanismos, ya sea aplicando criptografía simétrica o asimétrica. En ambos casos se hacen uso de las funciones resumen, o hash (Lucena López, 2011).

La autenticidad del mensaje le permite al receptor verificar que el mismo haya sido generado por quien dice ser, y que no se trata de un mensaje enviado por un tercero. . Por su parte, la integridad permite verificar que los

mensajes recibidos no hayan sido adulterados desde el momento en que se enviaron, independientemente de si la modificación en los mensajes se debió a errores propios del canal de comunicación, o a cambios intencionales introducidos por un atacante.

En criptografía simétrica la autenticidad se logra mediante un HMAC (Hash-based Message Authentication Code, Código de Autenticación Basado en Hash). El HMAC es el resultado de una función hash aplicada al mensaje original concatenado con una clave simétrica de autenticación. En otras palabras, se toma el mensaje original, se le concatena una clave simétrica, y sobre el conjunto se calcula una función hash. Luego el emisor envía el mensaje original y el hash calculado. El receptor deberá contar con la misma clave simétrica para verificar el hash del mensaje recibido. El receptor, al poder verificar el HMAC, podrá corroborar que el emisor es quien dice ser, y también estará constatando que el mensaje no ha sido modificado desde que se emitió, por lo que la integridad también quedará validada.

En criptografía asimétrica la autenticación de los mensajes se lleva a cabo mediante firmas digitales. La firma digital se calcula cifrando, con la clave privada del emisor, un hash calculado sobre el mensaje. El destinatario podría verificar la firma descifrando el hash con la clave pública del emisor, y comparando el hash obtenido con el que puede calcular del mensaje recibido. Al igual que en el caso de la autenticación por HMAC, el receptor que pueda verificar una firma digital de un mensaje estará constatando también que el mismo no haya sido adulterado durante la transmisión, por lo que validará también su integridad.

Un servicio relacionado con la autenticidad es el de no repudio. Si el destinatario puede verificar la autenticidad de un mensaje recibido, implica que el emisor del mismo no puede negar su envío.

II.2.3 Intercambio de claves

Cuando la seguridad de las comunicaciones en red se basa en claves simétricas de sesión, ambos nodos deben inicialmente compartir dicha clave a través de algún mecanismo que le impida a un atacante interceptar la conversación. Los nodos utilizarán dicha clave durante un tiempo configurado, y luego la descartarán y negociarán una nueva clave para la siguiente sesión. Uno de los métodos más utilizados para lograr este cometido es el protocolo de Diffie-Hellman (DH) (Rescorla, 1999). Este método permite a dos partes establecer una clave secreta compartida sobre un canal inseguro. Esta clave permitirá el cifrado y autenticación simétrica de los siguientes mensajes enviados entre los nodos. DH se usa en gran variedad de mecanismos de seguridad, entre ellos SSL/TLS, objeto de estudio de esta tesis.

Por otro lado, las claves de sesión generadas por DH son totalmente independientes unas de otras, por lo que un atacante que logre comprometer una de ellas no podrá usar dicha información para obtener la de la siguiente sesión. Esta característica se denomina PFS (Perfect Forward Secrecy - Secreto Perfecto Hacia Adelante) (Menezes et al., 2018).

DH permite a dos extremos de una comunicación obtener un número secreto común, que será utilizado como clave, sin ningún intercambio previo. Supóngase que dos extremos de una comunicación, llamados Alice y Bob, necesitan encontrar una clave simétrica común teniendo en cuenta que un atacante, Eva, espía el canal público. Para ello, y como se ve en la Fig. 2:

1. Alice y Bob eligen un número privado cada uno: a y b respectivamente.
2. Alice selecciona dos números públicos, g y p, primos.
3. Alice calcula un nuevo valor público: $A = g^a \ mod \ p$.
4. Alice envía a Bob los valores A, g y p.
5. Bob con esos datos puede calcular otro número público: $B = g^b \ mod \ p$.
6. Bob envia a Alice su resultado B.

7. Ambos nodos ahora pueden calcular la clave secreta K:
   - Alice calcula la clave con: $K = B^a \ mod \ p$
   - Bob calcula la clave con : $K = A^b \ mod \ p$

De esta forma un atacante que haya estado observando el canal público desde el inicio de la comunicación conocerá los valores g, p A y B, y aún así no podrá calcular K en un tiempo relativamente corto. Aquí:

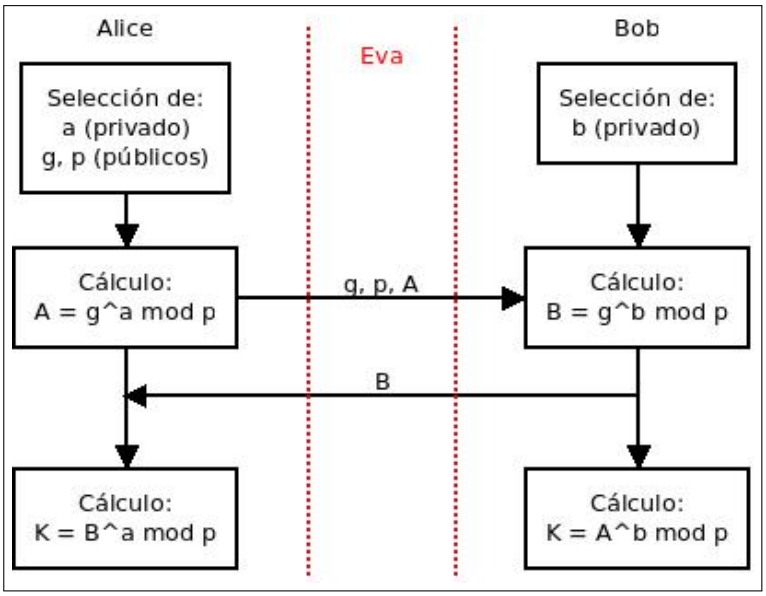

Figura 2: Intercambio Diffie-Hellman.
Fuente: (Lucena López, 2011)

- g se denomina base pública, es conocida por Alice, Bob y Eva.
- p se denomina módulo, es conocido por Alice, Bob y Eva.
- a es la clave privada de Alice, conocida únicamente por Alice.
- b es la clave privada de Bob, conocida únicamente por Bob.
- A es la clave pública de Alice, conocida por Alice, Bob y Eva.
- B es la clave pública de Bob, conocida por Alice, Bob y Eva.

Existen algunas variantes de DH como DH de curva elíptica (ECDH, por sus siglas en inglés), que combina el protocolo original con Criptografía de curva elíptica (ECC, por sus siglas en inglés) (Koblitz, 1987), una variante

de la criptografía asimétrica que hace uso de matemáticas de curva elíptica para proveer a los algoritmos una manera de reducir el tamaño de las claves, y con ello también el tiempo de procesamiento, sin perder el nivel de seguridad provisto por el protocolo original.

II.2.4 KEM: Key Encapsulation Mechanism

Este apartado introduce los conceptos básicos de mecanismos de encapsulamiento de claves dado que su entendimiento es necesario para comprender el lugar que ocupan los algoritmos post-cuánticos en algunas implementaciones analizadas en la parte DESARROLLO de esta tesis.

Los cifradores asimétricos como RSA suelen ser lentos y consumir más recursos que alternativas simétricas como AES (Donta, 2007), y es por esta razón que no suelen utilizarse para encriptar contenido de tráfico de red, como es el caso de TLS. Los cifradores asimétrico en TLS se utilizan para proteger el intercambio de claves simétricas que luego se usarán para encriptar el contenido del tráfico. Algoritmos como RSA encriptan bloques de tamaño equivalente al de la clave utilizada, es decir, si se usa RSA con claves de 2048 bits, los bloques de texto plano a cifrar serán de ese tamaño, 256 bytes. Ahora bien, el tamaño de clave más común en AES es de 256 bits (32 bytes), por lo que para encriptar una clave AES se requiere incorporar un relleno, o *padding*.

Hacer uso de rellenos en mecanismos criptográficos se considera inseguro ya que incrementa los vectores de ataque sobre la protección criptográfica (Bleichenbacher, 1998; Böck et al., 2018; Sullivan, 2016; Ronen et al., 2019; Manger, 2001). Para evitar utilizar el relleno generalmente se hace uso de mecanismos de encapsulamiento de claves (KEM, por sus siglas en inglés). Estos mecanismos facilitan la generación de claves simétricas aleatorias a partir de un sistema de clave pública y el uso de funciones hash.

En lugar de generar una clave secreta simétrica AES y luego encriptarla con un algoritmo asimétrico como RSA añadiendo el relleno, se define una API a la que se llama utilizando la clave pública del destinatario. La API retorna una clave simétrica AES encriptada lista para utilizar. Suponiendo que Alice es el emisor del mensaje y Bob el receptor, quien dispone de su clave pública, la secuencia para enviar un mensaje cifrado utilizando un mecanismo KEM es la siguiente:

1. Bob le envía su clave pública RSA a Alice. Dicha clave pública incluye un módulo propio del cifrador RSA.
2. Alice genera un número aleatorio entre 2 y el módulo de la clave pública de Bob menos uno.
3. Alice encripta dicho número con la clave pública de Bob. Este número aleatorio tiene el mismo tamaño que el módulo de la clave pública de Bob, por lo que no se añade relleno. A este valor cifrado se lo denomina clave encapsulada.
4. Alice genera una clave AES calculando un hash del número aleatorio.
5. Alice encripta el mensaje utilizando dicha clave AES.
6. Alice envía a Bob el mensaje cifrado y la clave encapsulada cifrada.
7. Bob desencripta la clave encapsulada utilizando su clave privada, y obtiene el número aleatorio generado por Alice.
8. Bob calcula el mismo hash que calculó Alice sobre este número aleatorio, obteniendo la clave AES.
9. Bob desencripta el mensaje utilizando esta clave AES.

El Alg. 1, adaptado de (Madden, 2021), muestra un ejemplo de las funciones necesarias para lograr el cifrado basado en KEM. Aquí:

- `rsa_encapsulate`: genera una clave AES de único uso y la retorna en texto plano y cifrada con una clave pública RSA.

- `rsa_decapsulate`: retorna la clave AES de único uso en texto plano descifrándola con la clave privada RSA.
- `encrypt`: encripta un mensaje usando la clave AES de único uso.
- `decrypt`: descifra un mensaje cifrado con la clave AES de único uso.

Este esquema permite la transmisión de mensajes cifrados utilizando un algoritmo simétrico como AES, cuya clave se comparte de manera segura mediante cifrado asimétrico (RSA en este ejemplo). Además el mecanismo permite cifrar cada bloque de mensaje utilizando claves únicas seleccionadas de manera aleatoria.

Otra ventaja que brinda un esquema KEM es la posibilidad de utilizar otros mecanismos asimétricos para realizar el encapsulamiento, no solamente RSA. (Madden, 2021) menciona un ejemplo basado en ECDH.

```
def rsa_encapsulate(publicKey):
  # random number in range(2,publicKey.modulus-1)
  random = generate_secure_random(publicKey);
  encryptedKey = rsa_encrypt(random, publicKey);
  aesKey = sha256(random);
  return (aesKey, encryptedKey)

def rsa_decapsulate(encryptedKey, privateKey):
  random = rsa_decrypt(encryptedKey, privateKey);
  return sha256(random)

def encrypt(message, publicKey):
  (aesKey, encryptedKey) = rsa_encapsulate(publicKey);
  ciphertext = aes_encrypt(aesKey, message);
  return (encryptedKey, ciphertext)

def decrypt(ciphertext, encryptedKey, privateKey):
  aesKey = rsa_decapsulate(encryptedKey, privateKey);
  message = aes_decrypt(aesKey, ciphertext);
  return message;
```

Algoritmo 1: Ejemplo de funciones necesarias para KEM.

III SSL/TLS

Como se mencionó en la Sección I, INTRODUCCIÓN, la seguridad de los protocolos de Aplicación es delegada, en general, a una subcapa en el stack TCP/IP denominada SSL/TLS. Esto hace que los protocolos de Aplicación sean agnósticos de los mecanismos de seguridad que usan, lo que facilita la implementación de nuevos algoritmos criptográficos.

Inicialmente los primeros desarrollos tendientes a proporcionar una capa de seguridad para los protocolos de aplicación los realizó Netscape en 1995, año en el que presentó SSLv2.0, la primer versión abierta del protocolo SSL, ya que la v1.0 nunca se liberó públicamente. La v2.0 contenía tantos fallos que obligó a realizar un rediseño completo del protocolo y lanzarlo como SSL v3.0. En 1999 se lanzó una nueva suite de protocolos: TLS v1.0. Si bien TLS es una actualización a la v3.0 de SSL, y no incluía grandes modificaciones, cambió de nombre y de versión dado que dichas modificaciones generaron incompatibilidades entre ambos protocolos.

Es importante aclarar en este punto el significado del término *cipher suite* en SSL/TLS. Este término hace referencia a un conjunto de algoritmos utilizados para darle seguridad a las comunicaciones:

- Un algoritmo de intercambio de claves.
- Uno o más algoritmos de cifrado, encargados de brindar confidencialidad a las comunicaciones.
- Un HMAC destinado a verificar la integridad de los datos.
- Puede incluir también un algoritmo de autenticación y firma digital para verificar la autenticidad de los mensajes.

Un ejemplo clásico de cipher suite podría ser:

```
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
```

Esta cipher suite tiene los siguientes componentes:

- `TLS`: define el protocolo de la cipher suite, generalmente será TLS.
- `ECDHE_RSA`: indica el algoritmo de intercambio de claves utilizado.
- `AES_128_GCM`: indica el cifrador a utilizar para encriptar los mensajes, el tamaño de la clave y el modo de operación.
- `SHA256`: indica el algoritmo HMAC usado.

TLS pasó por tres versiones que incluyeron mejoras progresivas en el rendimiento y la gestión de seguridad. Las versiones más comunes en la actualidad son las dos últimas, 1.2 y 1.3. Al momento de redactar este trabajo la versión 1.3 de TLS es considerada segura, mientras que versiones anteriores de TLS se consideran seguras dependiendo de los algoritmos de cifrado y autenticación que negocien cliente y servidor. Por su parte todas las versiones de SSL son consideradas inseguras (Möller et al., 2014).

Durante el año 2017 los principales navegadores web, Mozilla Firefox y Google Chrome, comenzaron a soportar la v1.3 de TLS de manera predeterminada. El mismo año wolfSSL añadió soporte a dicha versión en su implementación, y un año más tarde OpenSSL hizo lo mismo al lanzar su v1.1.1.

Con la intención de entender la secuencia de negociación de algoritmos, y posteriormente analizar el uso de criptografía post-cuántica en dicho intercambio, a continuación se describirán brevemente los mensajes intercambiados en TLS v1.2 y v1.3.

III.1 Negociación TLS v1.2 y v1.3

El motivo de los cambios que se produjeron desde la v1.2 a la v1.3 de TLS, además de incrementar el nivel de seguridad de las comunicaciones, fue

reducir el tiempo de establecimiento de la conexión, incrementando de esta forma el rendimiento en las comunicaciones seguras.

Tanto en la v1.2 como en la v1.3 de TLS la negociación de seguridad puede realizarse de dos maneras: con autenticación única del lado del servidor, o con autenticación mutua, que requiere algunos mensajes adicionales. En este apartado se analizará el intercambio de mensajes con autenticación mutua en ambas versiones de TLS.

III.1.1 TLS v1.2

En TLS v1.2 los pasos del *handshake*, o secuencia de mensajes enviados para establecer la comunicación, resumida en la Fig. 3, es la siguiente:

1. `Client Hello`: El cliente inicia la comunicación enviando al servidor la versión del protocolo, y los listados de cipher suites, de métodos de compresión soportados, y de extensiones habilitadas.
2. `Server Hello`: El servidor responde con su versión de protocolo, la cipher suite y método de compresión seleccionados, las extensiones soportadas, añade un número aleatorio y un identificador de sesión.
3. `Server Certificate`: El servidor le envía al cliente un certificado digital firmado que incluye su hostname y su la clave pública.
4. `Server key exchange generation`: El servidor genera un número público y uno privado para el intercambio Diffie-Hellman. En general se utiliza ECDH con curva x25519.
5. `Server Key Exchange`: El servidor le envía al cliente su número público y el valor público generado en base a Diffie-Hellman.
6. `Server Hello Done`: el servidor finaliza su parte del intercambio.
7. `Client key exchange generation`: el cliente genera un número público y uno privado para el intercambio Diffie-Hellman.
8. `Client Key Exchange`: El cliente le envía al servidor su número público y el resultado de calcular su parte de Diffie-Hellman.

9. `Client Encryption Keys Calculation`: El cliente genera ahora una clave de cifrado simétrico de DH que se utilizará en ambos extremos.
10. `Client Change Cipher Spec`: El cliente le indica al servidor que todos los mensajes siguientes serán cifrados.
11. `Client Handshake Finished`: Para verificar que la clave generada por ambos extremos es correcta el cliente crea un dato de verificación, lo cifra con dicha clave, y lo envía al servidor. Este valor se calcula como un hash de todos los mensajes intercambiados en la negociación, y sirve también para verificar la integridad de los mismos.
12. `Server Encryption Key Calculation`: El servidor calcula la clave DH.
13. `Server Change Cipher Spec`: El servidor indica que a partir de este momento todos sus mensajes serán cifrados.
14. `Server Handshake Finished`: El servidor calcula un dato de verificación tal y como hizo el cliente anteriormente, y lo envía al cliente.
15. Finalmente los nodos pueden comenzar a intercambiar datos cifrados.

III.1.2 TLS v1.3

La versión 1.3 de TLS reduce la cantidad de mensajes acelerando el proceso de conexión y proveyendo ventajas en seguridad. Los pasos en el intercambio de TLS v1.3, resumidos en la Fig. 4, son los siguientes:

1. `Client Hello`: El cliente envía la lista de cipher suites soportadas, el protocolo de intercambio de claves sugerido (variante DH), y su número público para dicho protocolo.
2. `Server Hello`: El servidor responde con el protocolo DH seleccionado, el resultado de su cálculo DH, y el mensaje `Server Finished`, similar a`Server Handshake Finished` de TLS 1.2. El servidor puede incluir opcionalmente su certificado digital.

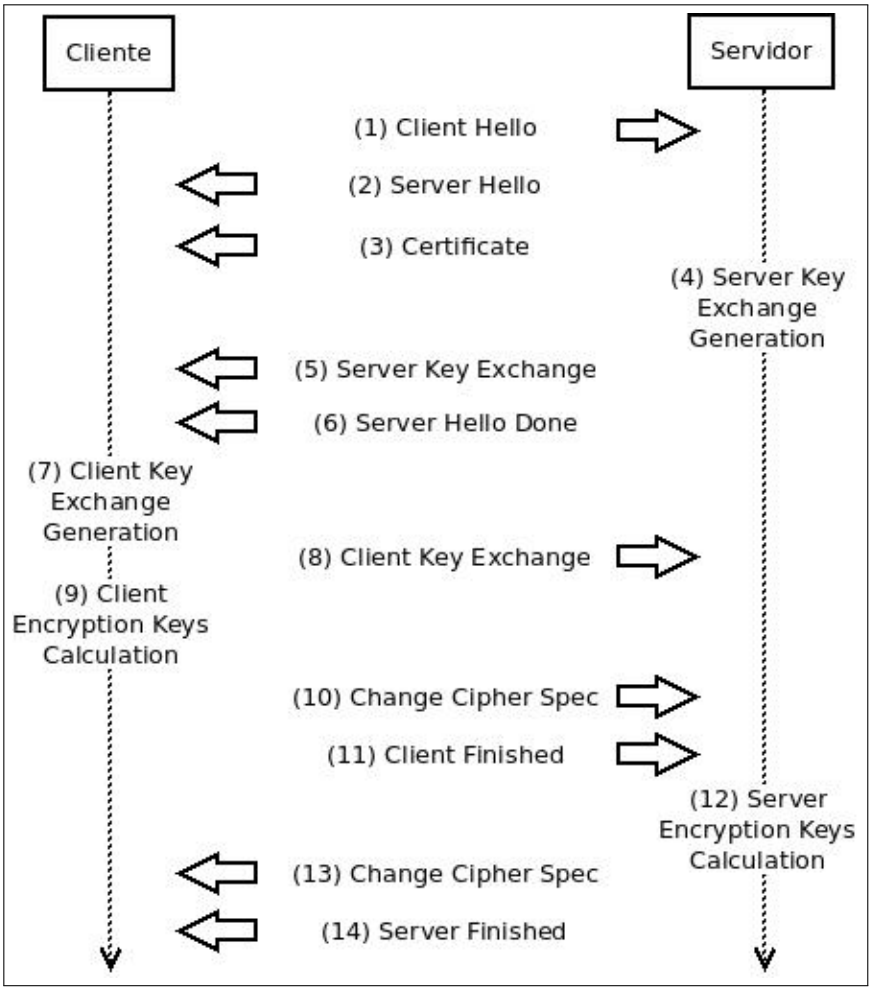

Figura 3: Handshake TLS v1.2.
Fuente: (Dierks y Rescorla, 2008)

3. Finalmente el cliente verifica, si es el caso, la firma del certificado del servidor, genera su clave DH, y envía el mensaje `Client Finished` al servidor. A partir de este paso, toda la comunicación será cifrada.

III.1.3 Mejoras de TLS v1.3

Como se vio en el detalle del handshake anterior, TLS v1.2 requiere más mensajes en el intercambio entre cliente y servidor, lo que incrementa el *round-trip time* en la comunicación. Considerando que no se pueden transmitir datos cifrados hasta que el handshake no se complete, este tiempo

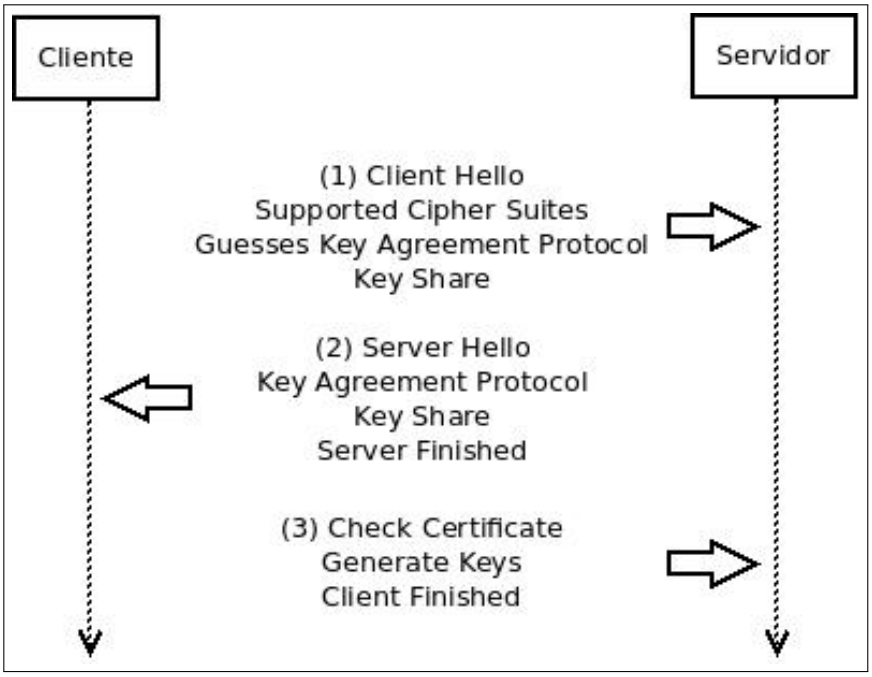

Figura 4: Handshake TLS v1.3.
Fuente: (Rescorla y Dierks, 2018)

puede reducir el rendimiento general de la comunicación. Además, desde el tercer mensaje del handshake todos los datos transmitidos viajan cifrados, lo que oculta mayor cantidad de información a un atacante.

Otro punto importante a favor de TLS v1.3 es la reducción de las cipher suites soportadas respecto de las que aceptaba la v1.2. Esto obliga a desactivar cipher suites obsoletas y posiblemente vulnerables. Además se reducen varios pasos en la negociación al permitir al cliente enviar sus números DH para la cipher suite que espera que sea aceptada por el servidor.

Reanudación de conexiones

Un factor importante en la mejora del rendimiento es el tiempo necesario para que el cliente se reconecte con un servidor al que ha estado conectado recientemente. En TLS v1.2 cuando un cliente vuelve a establecer una conexión con un servidor luego de un período de inactividad, enviará su identificador de sesión junto con el mensaje `Client Hello`, lo que le

permitirá al servidor detectar que dicho cliente ya ha estado conectado con anterioridad, y reanudará la comunicación con las claves previas enviando directamente el mensaje `Server Handshake Finished`. A este tipo de reinicio de conexión se lo denomina 1-RTT (1-Round Trip Time). Obviamente, al utilizar la misma clave de una sesión anterior no se garantiza el PFS.

TLS v1.3 incorpora una opción adicional para acelerar las reconexiones, denominada *Zero Round Trip Time*, o 0-RTT. Durante el handshake cliente y servidor negocian una clave pre-compartida PSK (Pre-Shared Key) a modo de clave de reconección. Durante la negociación inicial el cliente envía, mediante una extensión TLS, una versión cifrada de la PSK denominada `Session ticket`. Cuando un cliente se reconecta incluye el `Session ticket` en el mensaje `Client Hello`, junto con datos de aplicación cifrados con la PSK asociada. El servidor obtiene la PSK desde el `Session ticket` y descifra el mensaje.

Debido a que esta PSK no es obtenido en una ronda de Diffie-Hellman, no provee PFS si se compromete el ticket de sesión. En este caso un atacante podría descifrar la PSK y con él los siguientes datos del intercambio TLS. Por esta razón resulta necesario realizar un nuevo intercambio DH periódicamente para regenerar las claves de sesión. Aún así el tiempo de reconexión de TLS 1.3 es menor al de TLS 1.2 debido a que permite enviar datos de aplicación durante los mensajes de regeneración de la clave DH.

III.2 QSH: Quantum Safe Hybrid

En la Sección III.1 se analizó el intercambio de mensajes durante la negociación TLS en las versiones 1.2 y 1.3. En este apartado se describe la incorporación de QSH en la negociación de TLS v1.2.

QSH, o Quantum Safe Hybrid, es una cipher suite que provee un diseño modular para criptografía post-cuántica en TLS. Esta cipher suite se en-

cuentra en proceso de estandarización para TLS1.2 (Whyte et al., 2017a) y como algoritmo de intercambio de clave para TLS1.3 (Whyte et al., 2017b).

El agregado de QSH en la negociación de TLS 1.2 tiene un impacto directo en los mensajes `ClientHello`, `ServerHello`, `ServerKeyExchange`, y `ClientKeyExchange`. Al igual que ocurre en el intercambio TLS 1.2, el mensaje `ClientHello` incluye la lista de las cipher suites clásicas que el cliente ofrece para la negociación, a las que agrega una nueva, identificada como `TLS_QSH`. Los datos adicionales añadidos por QSH se resumen en la Fig. 5.

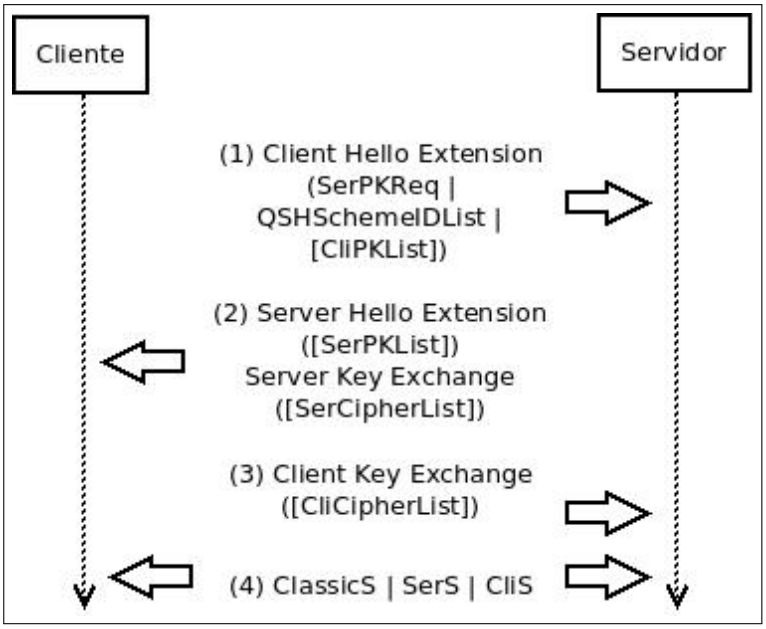

Figura 5: QSH: Dados añadidos al handshake TLS v1.2.
Fuente: (Whyte et al., 2017a)

El campo `ClientHelloExtension` puede agregar algunos datos adicionales tales como:

- `SerPKReq`: Una solicitud de la clave pública del servidor, e indica el número de claves públicas que el cliente espera recibir.
- `QSHSchemeIDList`: Una lista de identificadores de los esquemas criptográficos QSH soportados por el cliente.

- `CliPKList`: Una lista de las claves públicas del cliente correspondientes a los esquemas `QSHScheme` de la lista `QSHSchemeIDList`. No necesariamente debe proveer claves para todos los esquemas de la lista.

El mensaje `ServerHelloExtension` debe contener una lista de las claves públicas del servidor, `SerPKList`, correspondientes a cada `QSHScheme` de la lista `QSHSchemeIDList`. El mensaje `ServerKeyExchange` debe contener una lista de textos cifrados llamada `SerCipherList`, que incluye el campo `SerS`, el material secreto de clave del servidor. Este valor `SerS` viaja cifrado con la o las claves públicas del cliente, extraídas desde `CliPKList`.

Si el servidor envía el campo `SerPKList`, el cliente debe agregar en el mensaje `ClientKeyExchange` la lista de textos cifrados `CliCipherList`, que incluye al campo `CliS`, el material de clave secreta del cliente, cifrado a su vez con la o las claves públicas del servidor, obtenidas desde `SerPKList`.

El parámetro final negociado entre cliente y servidor es una concatenación de los materiales de clave previamente intercambiados, `SerS` y `CliS`, en ese orden, por lo que `SerS` y `CliS` no pueden ser nulos. A esta concatenación se la denomina `premaster`. Al final del handshake, ambos nodos generan una clave secreta basada en el `premaster` y utilizando una función pseudo-aleatoria con suficiente entropía. La seguridad radica en que un atacante con una computadora cuántica no conozca los valores `SerS` y `CliS`, por lo que tampoco podrá conocer el `premaster` final. A su vez, también se garantiza que el `premaster` permanecerá en secreto incluso si las claves de sesión del cliente y/o del servidor se ven comprometidas.

Cabe mencionar algunos detalles de la estructura de datos que define los identificadores de los esquemas criptográficos post-cuánticos soportados por QSH: `QSHSchemeId` (véase Alg. 2). Si bien contiene códigos reservados para uso futuro (`reserved`), ya se disponen de algunos esquemas fijos de cifrado resistentes al criptoanálisis cuántico.

```
enum {
  ntru_eess439 (0x0101),
  ntru_eess593 (0x0102),
  ntru_eess743 (0x0103),
  reserved     (0x0102..0x01FF),
  lwe_XXX      (0x0201),
  reserved     (0x0202..0x02FF),
  hfe_XXX      (0x0301),
  reserved     (0x0302..0x03FF),
  reserved     (0x0400..0xFEFF),
  (0xFFFF)
} QSHSchemeID;
```

Algoritmo 2: Estructura de datos QSHSchemeId.

Como puede observarse, los principales esquemas definidos hacen referencia a algunos de los algoritmos utilizados en el presente trabajo, tales como NTRUEncrypt con diferentes parámetros, o variantes del algoritmo de aprendizaje con errores (LWE, por sus siglas en inglés).

Si bien esta aproximación a una cipher suite post-cuántica es válida únicamente para TLS v1.2, en TLS v1.3 también existen proyectos de inclusión de un intercambio de claves post-cuántico. En TLS 1.3 el protocolo no se modifica como ocurre en el caso anterior, sino que se consideran los nuevos mecanismos de intercambio de claves como si se tratase de grupos ECDH adicionales, con la posibilidad de negociar varios intercambios de clave simultáneamente, y luego combinar los resultados de cada uno de ellos mediante el uso de una función de derivación de clave. De esta manera se considera segura la clave negociada siempre que uno de los mecanismos utilizados para el intercambio sea seguro. Así, el nuevo intercambio de claves en TLS 1.3 podría ser tanto un clásico DH o ECDH, como un intercambio híbrido.

Finalmente, cabe resaltar que en TLS 1.3 no se incluye una cipher suite específica para QSH como ocurría con `TLS_QSH_*` en la TLS 1.2, sino que los mecanismos de resistencia cuántica se ejecutan durante la etapa de intercambio de claves original de TLS.

IV COMPUTACIÓN CUÁNTICA Y SEGURIDAD INFORMÁTICA

En este capítulo se realizará una breve introducción a los conceptos fundamentales de la computación y la información cuánticas, como así también a los principales algoritmos cuánticos, importante motivación para el desarrollo de los temas tratados en este trabajo. Se introducirá la transformada cuántica de Fourier, los algoritmos de Shor y de Grover, y cómo éstos afectan a los mecanismos criptográficos actuales.

Las computadoras actuales funcionan convirtiendo información en series de dígitos binarios, o bits, y operando con estos bits mediante el uso de circuitos integrados que contienen millones de transistores. Cada bit tiene sólo dos valores posibles, 0 y 1. Mediante la manipulación de estos bits las computadoras pueden realizar todo su procesamiento.

Una computadora cuántica también representa información en bits, llamados bits cuánticos, o *qubits*. Al igual que un bit clásico, un qubit puede tener un valor de 0 o 1, pero a diferencia de ellos, que solamente pueden estar en uno de estos dos estados, un qubit también puede estar en un estado especial denominado *superposición*, en el que vale 0 y 1 al mismo tiempo. Cuando se extiende el sistema a una gran cantidad de qubits, la superposición de estados le da a la computadora cuántica un enorme potencial.

IV.1 FUNDAMENTOS DE LA FÍSICA CUÁNTICA

A principios del siglo XX la mecánica cuántica fue uno de los modelos mejor conocidos para explicar el mundo de la física. Esta teoría describe el comportamiento de las partículas en escalas de energía y distancia muy pequeñas, sentando las bases para entender las propiedades de la materia.

Un objeto cuántico no existe en un estado determinado y visible. Cada vez que se observa un objeto cuántico se ve como una partícula, pero cuando no está siendo observado se comporta como una onda. Es por esto que los fenómenos físicos en este sentido tienen una dualidad onda-partícula. Mientras que la evolución del sistema tiene un comportamiento ondulatorio, cualquier medición del mismo obtendrá un valor que podría demostrar que se trata de una partícula. Un ejemplo clásico de este comportamiento es el experimento de la doble ranura (Fig. 6).

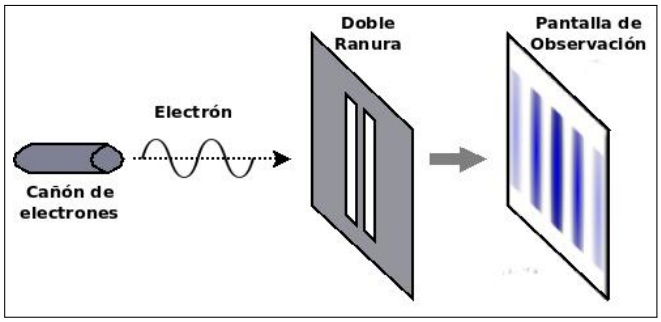

Figura 6: Experimento de la doble ranura.
Fuente: (Kalliauer, 2017)

Un objeto cuántico puede existir en múltiples estados a la vez, con cada uno de ellos coexistiendo e interfiriéndose como si fueran ondas. El estado de cualquier sistema cuántico se describe en términos de funciones de onda. En muchos casos el estado de un sistema puede ser expresado matemáticamente como la suma de los posibles sub-estados que lo constituyen, escalados por un número complejo que representa el peso relativo de cada uno. Este valor complejo permite expresar, mediante parte real e imaginaria, la amplitud y la fase de la onda. Por ejemplo, $Ae^{i\theta}$ hace referencia a una función de onda con amplitud A y fase θ. Se dice que los estados son *coherentes* porque cada uno puede interferir con los otros en forma constructiva y destructiva, tal y como ocurre con las ondas.

Cuando se intenta observar un sistema cuántico se ve sólo uno de los estados componentes, y esto ocurre con una probabilidad proporcional al cua-

drado del valor absoluto de su coeficiente. Para un observador el sistema siempre parecerá clásico cuando se lo mide. La observación de un objeto cuántico o de un sistema cuántico (conjunto de objetos cuánticos) ocurre cuando el objeto interactúa con un sistema físico mayor que extrae información de él. Así, el hecho de medir el objeto cuántico tiene un efecto disruptivo sobre su estado, el aspecto de la onda que fue medida representa sólo uno de los estados observables, con la consecuente pérdida de información. Luego de la medición, la función de onda del objeto cuántico medido será el resultado de la medición, y perderá su estado previo.

Para visualizar esta situación considérese el ejemplo de una moneda sobre una mesa. En el mundo de la física clásica el estado de la moneda será *cara* o *cruz*. Una versión cuántica de la moneda podría existir en una combinación o superposición de ambos estados. La función de onda de la moneda cuántica podría ser escrita como la suma ponderada de ambos estados, escalada por los coeficientes C_C y C_Z, donde C y Z representan el cara y cruz de la moneda respectivamente. Un intento por observar el estado de la moneda cuántica conducirá a encontrarla en uno de sus dos estados, cara o cruz, con una probabilidad proporcional al cuadrado de su coeficiente. A esta medición, que deriva en la configuración de un estado particular del objeto cuántico, se la suele llamar “colapso de la función de onda”.

Además, bajo algunas circunstancias, dos o mas objetos cuánticos en un sistema pueden estar intrínsecamente enlazados, de modo que la medición de uno de ellos revelará también el estado del otro, haciéndo colapsar su función de onda, independientemente de la distancia que los separe. Esta propiedad se conoce como entrelazamiento cuántico, y es la clave principal del gran potencial de la computación cuántica. Esto ocurre cuando las funciones de onda de ambas partículas no son separables matemáticamente, es decir, cuando la función de onda del sistema completo no puede escribir-

se como un producto de las funciones de onda de cada partícula individual. Para este fenómeno no existe un equivalente en la física clásica.

IV.2 INFORMACIÓN CUÁNTICA

La ciencia de la información cuántica explora la manera en la que se puede codificar la información en un sistema cuántico. En este campo de estudio son de interés tres ramas importantes: las comunicaciones cuánticas, la detección cuántica y la computación cuántica.

Las investigación y desarrollo en **comunicaciones cuánticas** se centran en el transporte o intercambio de información codificándola en un sistema cuántico. Es probable que se necesiten nuevos protocolos de comunicación para que las computadoras cuánticas puedan transportar información. Un campo importante en este punto es el de la criptografía cuántica, que diseña sistemas de comunicaciones basados en principios de física cuántica.

Un uso interesante en este campo es el de QKD (Quantum Key Distribution - Distribución de claves cuánticas), un método de distribución de claves criptográficas llevado a cabo por medio de mecanismos cuánticos. El protocolo mejor conocido en este entorno es el BB84, desarrollado por Charlie Bennet y Gilles Brasard en 1984 (Bennett et al., 1992). Este protocolo fue desarrollado experimentalmente en cables de fibra óptica y comunicaciones satelitales, y ha sido la guía de varios desarrollos comerciales.

Por su parte la **detección cuántica** involucra el estudio y desarrollo de sistemas cuánticos que son extremadamente sensibles al entorno, y pueden ser explotados para medir propiedades físicas tales como campos magnéticos, eléctricos, gravitacionales, o incluso temperatura, con una precisión mayor que la que ofrecen los las tecnologías actuales.

Finalmente, la **computación cuántica** implica el estudio de propiedades cuánticas para ejecutar cálculos computacionales. Una computadora cuán-

tica es un sistema físico compuesto por una colección de qubits apareados. Estos qubits pueden ser manipulados para que la medición del estado final del sistema pueda resolver un problema con una muy alta probabilidad de éxito. Los qubits de una computadora cuántica deben estar lo suficientemente aislados del entorno para que su estado se mantenga coherente durante el tiempo que lleve realizar los cálculos. El campo de la computación está entrando en la era NISQ (Noisy Intermediate Scale Quantum - Dispositivos Cuánticos de Escala Intermedia) (Preskill, 2018). Esto es, la construcción de computadoras cuánticas lo suficientemente grandes (decenas de cientos de qubits) que no pueden ser simulados eficientemente en computadoras clásicas. Aún no se logra construir un procesador cuántico que gestione correctamente el ruido, por lo que los algoritmos cuánticos no pueden implementarse directamente.

IV.3 ALGORITMOS CUÁNTICOS Y SEGURIDAD

En computación clásica se verifica que el número total de operaciones requeridas para resolver un problema es independiente del diseño de la computadora que las ejecute. A este principio se lo denomina tesis de Church-Turing extendida, e implica que, para resolver un problema computacional de manera más rápida, se deben realizar algunas de estas tareas:

1. Reducir el tiempo en implementar cada operación simple.
2. Ejecutar muchas operaciones en paralelo.
3. Reducir el total de operaciones para completar el cálculo.

La computación cuántica viola la tesis extendida pudiendo resolver ciertos cálculos con menos operaciones que una computadora clásica que use el mejor algoritmo conocido para dicha tarea. Se abre así una nueva posibilidad para solucionar problemas computacionales. Esto genera gran preocupación en la comunidad de expertos en seguridad informática ya a que

la dificultad de resolver ciertos cálculos con una computadora clásica es la base de muchos sistemas criptográficos actuales.

Durante las últimas décadas se han estudiado intensamente los problemas de factorización de números grandes. En el caso de la factorización de un número N de n bits ($N = 2^n$), los posibles divisores primos de N incluyen todos los números primos menores que N, y la cantidad de cifras a evaluar es $\exp(n)$. Los algoritmos clásicos buscan los divisores primos de N con operaciones de $exp(O(n^{1/3}))$ pasos, mientras que el algoritmo cuántico de Shor (descrito en la Sección IV.3.2) lo hace sólo $O(n^3)$ pasos, y nuevas variantes mejoradas pueden resolverlo hasta en $O(n^2 \log[n])$ pasos.

Entre los objetivos de la teoría de algoritmos se encuentra resolver problemas de complejidad P, tareas computacionales en un número de pasos que escale polinómicamente según el tamaño de N. En computación cuántica la clase de complejidad que corresponde con estos problemas, y que contiene todas las tareas que una computadora cuántica puede resolver en tiempo polinomial cuántico de error acotado (BQP, por sus siglas en inglés).

Es importante notar que las computadoras cuánticas no aceleran uniformemente todos los problemas computacionales. Existen problemas como los NP, que no pueden resolverse en tiempo polinomial. Esta categoría incluye problemas aún más complejos, los NP-completos (Karp, 1975). El algoritmo de Shor prueba que los algoritmos cuánticos requieren un tiempo exponencial para resolver estos problemas NP-completos (Bennett et al., 1997).

El diseño de algoritmos cuánticos sigue principios diferentes de los que se utilizan en algoritmos clásicos. Los algoritmos que logran aprovechar un procesador cuántico utilizan paradigmas o técnicas de construcción que no tienen una contraparte en computación clásica.

IV.3.1 Transformada cuántica de Fourier

La transformada de Fourier es una operación que convierte la representación de una señal en otra forma de representación diferente. La transformada clásica convierte una función del dominio del tiempo al de la frecuencia. La función inversa, denominada anti-transformada de Fourier, permite convertir una función del dominio de la frecuencia al del tiempo sin pérdida de información. Esta propiedad es clave para cualquier operación en una computadora cuántica. La transformada cuántica de Fourier, o QFT (Quantum Fourier Transform) es el equivalente cuántico de la transformada clásica, y constituye un bloque básico de construcción de algoritmos cuánticos.

Debido a la utilidad de la transformada de Fourier muchos algoritmos la utilizan en computación clásica. Una de las mejores implementaciones es la Transformada Rápida de Fourier (FFT, por sus siglas en inglés), que se evalúa en un tiempo $O(N \log N)$ para un número N de n bits. Si bien la FFT es eficiente, existen implementaciones de la QFT que, haciendo uso de compuertas de uno o dos qubits, se evalúan en un tiempo $O(\log^2 N)$, que representa una aceleración exponencial respecto de la FFT (NAP, 2019). Cabe mencionar que esta eficiencia solamente se da cuando las entradas de la QFT son pre-codificadas en qubits, y no leídas directamente. De esta forma la QFT permite aprovechar las características de las computadoras cuánticas, y por ello es útil para crear una gran cantidad de algoritmos cuánticos, entre los que se encuentran la factorización, o la búsqueda de estructuras ocultas y estimación de fase cuántica.

IV.3.2 Factorización cuántica y el algoritmo de Shor

Peter Shor descubrió en 1994 algoritmos cuánticos de tiempo polinómico para la factorización y el calculo de logaritmos discretos (Shor, 1994). Esto representó un enorme avance en el campo de los algoritmos cuánticos por

la aparente aceleración en comparación con las variantes clásicas. Este tipo de algoritmos se consideran una buena forma de explotar la aceleración exponencial de la QFT, incluso teniendo en cuenta las limitaciones de entrada y salida del muestreo de Fourier.

Para poder aprovechar el poder de la QFT, Shor primero convierte el problema de encontrar los factores de un número en otro problema, hallar un patrón repetido. Shor fue capaz de mostrar que la factorización de un problema es equivalente a encontrar el período de una secuencia de números, aunque ésta sea exponencialmente más larga que la cantidad de bits del número a factorizar. En un ordenador clásico esta operación no mejoraría el tiempo de cálculo, ya que la computadora debería generar esta secuencia de 2^n números para factorizar un número de n bits, lo que llevaría a una cantidad de tiempo exponencial. Sin embargo, en un ordenador cuántico una secuencia exponencialmente larga puede codificarse sólo en n qubits, y generarse en un tiempo polinómico. Una vez que se ha generado esa secuencia, se puede usar la QFT para encontrar su período.

Debe tenerse en cuenta que el resultado devuelto será solamente una muestra de las amplitudes de la transformada de Fourier, aunque esto no es una limitante ya que, con una alta probabilidad, la información deseada será la que se encuentre en dicha muestra. De esta manera, si se desplegara el algoritmo de Shor en una computadora cuántica perfecta, sería posible calcular la clave secreta de la mayoría de los criptosistemas de clave pública tales como RSA o DSA, o algoritmos de intercambio Diffie-Hellman.

IV.3.3 Algoritmo de Grover

El algoritmo de Grover aborda el problema de encontrar las entradas únicas de una función que producirán una determinada salida (Grover, 1996). En computación clásica representa un problema clasificado como NP-duro, es decir, no tiene soluciones de tiempo polinómico conocidas. A falta de infor-

mación sobre la naturaleza de la función, el algoritmo clásico más rápido que se conoce para este problema es el de búsqueda exhaustiva, o la exploración de todas las entradas posibles hasta dar con la solución, proceso que toma $O(N) = O(2^n)$ pasos, donde n es el número de bits necesarios para representar la entrada. El algoritmo de Grover resuelve este problema en $O(N^{1/2})$ pasos. Aunque esto es sólo una aceleración polinómica sobre el mejor enfoque clásico, podría ser significativo en la práctica, y sería capaz comprometer algunas operaciones criptográficas.

IV.4 ALGORITMOS Y RESISTENCIA CUÁNTICA

En esta sección se analizarán las implicaciones de la computación cuántica en los mecanismos criptográficos utilizados en las comunicaciones: intercambio de claves, cifrado asimétrico, cifrado simétrico, firmas y certificados digitales, funciones resumen y contraseñas.

IV.4.1 Intercambio de claves y cifrado asimétrico

Un protocolo de intercambio de claves basado en Diffie-Hellman asume que ciertos problemas algebraicos son intratables, es decir, hasta la fecha no se ha encontrado un algoritmo que pueda resolverlo eficientemente. Para romper ECDH se necesita resolver un problema de logaritmos discretos de curva elíptica. Una computadora clásica puede resolver un problema de este tipo en un tiempo exponencial $2^{n/2}$, siendo n el tamaño de la clave. Así, por ejemplo, para una clave de 256 bits el mejor algoritmo conocido puede vulnerarla en un tiempo 2^{128}. Este tiempo es equivalente al requerido para atacar un cifrados simétrico AES-GCM, lo que nos da una idea de que el nivel de seguridad del intercambio de claves ECDH. Los algoritmos asimétricos como RSA o DSA también están basados en logaritmos discretos.

Ahora bien, los problemas de este tipo, si bien se consideran difíciles de resolver con computadoras clásicas, pueden ser vulnerados usando el algoritmo de Shor. De esta forma una computadora cuántica podría romper la mayoría de los algoritmos asimétricos y de intercambio de claves actuales. Por ejemplo, una computadora cuántica de 2300 qubits lógicos podría usar el algoritmo de Shor para romper RSA de 1024 bits en aproximadamente un día (NAP, 2019).

IV.4.2 Cifrado simétrico

Una vez que ambos extremos establecieron su clave secreta en TLS, implementan cifrado simétrico para asegurar la privacidad de la comunicación. Uno de los algoritmos criptográficos más utilizados es el Estándar de Cifrado Avanzado (AES, por sus siglas en inglés) (Daemen y Rijmen, 1999), y en general se hace uso del modo GCM. AES-GCM soporta claves de 128, 192 y 256 bits. Considerando que en los protocolos estándares en Internet los primeros bytes de los mensajes cifrados suelen ser constantes, un atacante que intercepte un mensaje cifrado con AES-GCM de n bits debería probar, por medio de fuerza bruta, las 2^n claves posibles para descifrar el mensaje. La búsqueda terminará cuando el atacante logre visualizar los primeros bytes conocidos del mensaje, y con ello ya podrá acceder al resto del texto plano original. Para el caso de claves de 192 o 256 bits se han descubierto ataques que permiten romper el algoritmo en 2^{176} y 2^{119} ciclos respectivamente (Biryukov et al., 2010). Aunque estos ataques resultan más rápidos que una búsqueda exhaustiva, igualmente son impracticables y no presentan una amenaza real al algoritmo.

El algoritmo de Grover puede identificar la clave secreta de AES-GCM de 128 bits en un tiempo proporcional a $2^{128/2} = 2^{64}$ corriendo en una computadora cuántica de alrededor de 3000 qubits lógicos (NAP, 2019). Resulta difícil calcular el tiempo real que le tomaría a una computadora cuántica ya

que cada paso del algoritmo de Grover debe descomponerse en primitivas simples. Teniendo en cuenta esto, (NAP, 2019) estima que un cluster de computadoras cuánticas podría romper AES-GCM de 128b en un mes. Incluso en el momento en que una computadora cuántica pueda romper AES-GCM en un tiempo relativamente corto, incrementar el tamaño de la clave haría muy difícil encontrar una solución usando Grover. De esta forma, AES-GCM puede ser muy fácilmente asegurado contra el criptoanálisis cuántico.

No obstante, AES-GCM está diseñado para resistir ataques sofisticados ejecutados por computadoras clásicas, como criptoanálisis lineal y diferencial, pero no está diseñado para ataques cuánticos sofisticados. Es decir, es posible que exista un algoritmo desconocido que permita encontrar una solución a AES-GCM en una computadora cuántica de manera más eficiente que Grover. En este escenario, incrementar el tamaño de la clave no asegura que AES-GCM se mantenga seguro.

IV.4.3 Firma digital y Certificados

Las firmas digitales son un mecanismo criptográfico utilizados para verificar la integridad y autenticidad de datos. Una CA (Certification Authority - Autoridad Certificante) firma un certificado digital para un individuo u organización, de modo que quien necesite verificar la identidad de dicha entidad pueda hacerlo en la medida en que confíe en la CA. Este es el uso que le da TLS a las firmas digitales. Los dos esquemas de firma digital más utilizados son RSA y ECDSA. La seguridad de RSA se basa en problemas de factorización, y la de ECDSA en logaritmos discretos, ambas técnicas vulnerables al algoritmo de Shor. Un atacante que tenga una computadora cuántica podría utilizar el algoritmo de Shor para romper las firmas RSA y ECDSA. El atacante podría falsificar certificados digitales, y con ello suplantar la identidad de una de las partes en una comunicación.

IV.4.4 Funciones resumen o hash

Algunas primitivas adicionales a tener en cuenta son las funciones hash o resumen. La función hash genera cadenas de tamaño fijo utilizando como entrada un contenido arbitrario. La seguridad de una función hash se basa en cálculos matemáticos unidireccionales, imposible de revertir. Además, una función hash debe ser resistente a las colisiones, es decir, debería ser muy difícil (si no imposible) encontrar dos mensajes de entrada que generen la misma cadena resumen de salida.

En principio es esperable que una función hash SHA256 no sea vulnerable al criptoanálisis cuántico. Se supone que es imposible romper un hash de ese tamaño en una computadora cuántica de aproximadamente 2400 qubits lógicos corriendo el algoritmo de Grover (NAP, 2019).

(NAP, 2019) menciona también que el algoritmo de Grover también puede utilizarse para realizar ataques de fuerza bruta a cadenas hash que representan contraseñas. Si bien este tipo de ataques no explotan una vulnerabilidad del algoritmo hash en si mismo, sí podrían romper la seguridad de un sistema al encontrar, mediante búsqueda exhaustiva, una cadena de texto que de como resultado un hash determinado. No obstante, los mecanismos de corrección de errores necesarios para implementar Grover extenderían el tiempo del ataque a cifras poco prácticas.

V CRIPTOGRAFÍA POST-CUÁNTICA

En este capítulo se partirá de las motivaciones que dan origen a la criptografía post-cuántica, y se detallarán algunas características de sus principales criptosistemas. Se mencionarán también algunos desafíos que tiene por delante la criptografía post-cuántica, y sus diferencias con su par cuántico. Finalmente se explicará brevemente el proceso de estandarización y los niveles de seguridad de los algoritmos, ya las implementaciones utilizadas para las pruebas hacen referencia a los mismos.

La seguridad de los algoritmos post-cuánticos descansa en problemas matemáticos que se cree que no pueden ser resueltos por computadoras cuánticas. Igualmente, y como en cualquier rama de la criptografía, debe tenerse en cuenta que la seguridad de estos algoritmos no puede establecerse contra técnicas criptoanalíticas hasta hoy desconocidas (NAP, 2019).

Como se vio en la Sección IV.3, la seguridad post-cuántica en los algoritmos de cifrado simétrico y de hash puede obtenerse aumentando el tamaño de la clave de cifrado o la salida de la función resumen respectivamente. En primera instancia algoritmos de cifrado como AES-GCM de 256 bits y hashes como SHA256 serían resistentes al criptoanálisis cuántico.

Por su parte, los algoritmos de cifrado asimétrico, de firma digital y los protocolos de intercambio de claves se consideran vulnerables a ataques cuánticos. Así, algoritmos como RSA, ECDH o ECDSA deberán ser actualizados a nuevos esquemas de cifrado que incorporen mecanismos post-cuánticos, o criptosistemas híbridos.

No hay duda de que el cifrado de clave pública es necesario para las comunicaciones en Internet, así como también es un hecho que estos criptosistemas pueden volverse inseguros ante el criptoanálisis cuántico (Schmidt,

2010). Si bien una computadora cuántica puede romper varios algoritmos actuales en un tiempo que resulte útil al atacante, esto no implica que la criptografía de clave pública se vuelva obsoleta.

V.1 CANDIDATOS POST-CUÁNTICOS

Existen varios sistemas criptográficos de clave pública resistentes a ataques cuánticos, candidatos prometedores que resultan de interés en el presente trabajo. Entre ellos se encuentran:

- Criptografía basada en hash
- Criptografía basada en código
- Criptografía basada en retículos
- Criptografía basada en ecuaciones cuadráticas multivariadas
- Criptografía basada en isogenias supersingulares
- Criptografía de clave secreta

V.1.1 Criptografía basada en hash

Los sistemas de firma digital basados en funciones hash que datan de los años ’80 se suponen seguros si se utilizan funciones hash consideradas invulnerables al ataque cuántico. El mayor problema de estos esquemas es que generan firmas de una longitud relativa grande, y por ello pueden ser utilizadas sólo en ciertos ambientes, como la firma de paquetes de software en gestores de aplicaciones.

Un ejemplo representativo es el sistema de firma de clave pública de Merkle (1979) construido sobre la idea de firma de un mensaje de Lamport y Diffie (Becker, 2008). Otra propuesta interesante es el esquema de firma LMSS (Leighton-Micali Signature Scheme, Esquema de Firma de Leighton-Micali) (McGrew et al., 2019).

V.1.2 Criptografía basada en código

La teoría de códigos es la ciencia que estudia los esquemas de codificación que permiten a dos partes comunicarse sobre un canal ruidoso. El emisor codifica un mensaje de modo que el receptor puede decodificarlo incluso si el canal presenta un ruido acotado. Los investigadores en esta rama se encuentran estudiando algunos esquemas de codificación que resultan difíciles de decodificar. De hecho, para ciertos esquemas el mejor algoritmo para decodificar requiere un tiempo exponencial en computadoras clásicas. Además, el problema de la decodificación parece ser difícil incluso para computadoras cuánticas, de modo que los criptógrafos están empleando este tipo de codificación para construir criptosistemas post-cuánticos.

Un ejemplo típico es el sistema de cifrado asimétrico de código Goppa oculto de McEliece (McEliece, 1978; Bernstein y Lange, 2008) que puede ser usado para realizar intercambio de claves post-cuántico. Recientemente surgieron algunas variantes prometedoras como CAKE (Barreto et al., 2017).

V.1.3 Criptografía basada en retículos

Un retículo o *lattice* es un conjunto parcialmente ordenado en el que, para cada par de elementos, existen un supremo y un ínfimo. Si se tiene un conjunto parcialmente ordenado L, se lo denomina retículo si tiene un supremo por pares, y tiene un ínfimo por pares, que también pertenece a L.

Uno de los problemas computacionales más conocidos en este aspecto es el de encontrar el vector más corto en una grilla. Todos los algoritmos clásicos actuales que resuelven este problema lo hacen en un tiempo exponencial respecto de la dimensión de la grilla, y existen evidencias de que también tomaría un tiempo exponencial en computación cuántica (Regev, 2009). Se han presentado varias propuestas en esta categoría, y si se veri-

ficara que una computadora cuántica no puede resolver de manera sencilla estos problemas, los algoritmos basados en retículos serían seguros.

El ejemplo que más relevancia tiene es el sistema de cifrado de clave pública Hoffstein-Pipher-Silverman “NTRU” (Hoffstein et al., 1998). Otros candidatos interesantes son New Hope (Alkim et al., 2016) y Frodo (Bos et al., 2016). Tanto Frodo como New Hope hacen uso de problemas computacionales de Aprendizaje con errores sobre anillos, o RLWE (Ring Learning with Errors, Anillo de Aprendizaje con Errores). Se consideran algoritmos post-cuánticos porque se presume que los problemas RLWE son difíciles de resolver, incluso para computadoras cuánticas (Lyubashevsky et al., 2012).

Google ha estado experimentando con New Hope en su navegador Chrome (Braithwaite, 2016). Uno de los resultados obtenidos arrojó que el incremento de tiempo en que se incurre al utilizar New Hope es despreciable y no resulta un impedimento para su implementación práctica.

Por otro lado NTRU es un criptosistema que consiste en dos algoritmos: NTRUEncrypt y NTRUSign para cifrado/descifrado y firma digital respectivamente. En 2017 se liberó bajo dominio público y puede ser utilizado en aplicaciones libres bajo licencia GPL. NTRU puede ejecutar operaciones con un nivel de seguridad similar al de RSA, pero con mayor eficiencia (Baktu, 2017). Esto es debido a que el tiempo que demora RSA en realizar las operaciones se incrementa con el cubo del tamaño de la clave, mientras que en NTRU el incremento de tiempo es cuadrático. Se han realizado experimentos con una GPU GTX280 y se pudo procesar NTRU con un nivel de seguridad de 256 bits a una tasa sólo 20 veces más lenta que la de un cifrador simétrico AES (Hermans et al., 2010). A diferencia de RSA y ECDSA, NTRU es resistente a las técnicas criptoanalíticas cuánticas conocidas. (Perlner y Cooper, 2009; Stehlé y Steinfeld, 2013) menciona a NTRU como una alternativa para cifrado y firma digital resistentes al algoritmo de Shor,

y hace alusión a que, de todos los algoritmos criptográficos basados en el esquema de retículo, NTRU parece ser el más práctico, incluso más que versiones alternativas como Stehle-Steinfeld NTRU.

V.1.4 Criptografía multivariable

La criptografía multivariable o basada en ecuaciones cuadráticas multivariable es un término genérico para funciones de criptografía asimétrica basada en ecuaciones polinomiales multivariable sobre un campo finito F. Si estos polinomios son de grado 2 hablamos de ecuaciones cuadráticas multivariable. Los problemas matemáticos basados en este tipo de ecuaciones son de tipo NP-completos (Hartmanis, 1982), característica que los convierte en candidatos post-cuánticos. La criptografía multivariable ha sido muy productiva en términos de diseño y criptoanálisis, sobre todo ahora que han pasado varios años de estudio y se consideran algoritmos estables y robustos. Este tipo de mecanismos permite generar firmas digitales más cortas que las proporcionadas por otros algoritmos post-cuánticos.

Un ejemplo relevante es el sistema de firma de clave pública basadas en ecuaciones de campo oculto (HFE, por sus siglas en inglés) (Patarin, 1996).

V.1.5 Criptografía basada en isogenias supersingulares

Los gráficos de isogenia supersingular son un tipo de gráficas de expansión provenientes de la teoría de números computacionales, y se ven aplicados comúnmente en la criptografía de curva elíptica. Se considera que los algoritmos basados en isogenia supersingular son post-cuánticos.

Un candidato reciente en esta categoría es Diffie-Hellman de isogenias supersingulares (SIDH, por sus siglas en inglés) (Costello et al., 2016), un algoritmo de intercambio de claves basado en Diffie-Hellman. SIDH genera menos tráfico de red que el algoritmo de retículos New Hope, y aunque requiere más tiempo de cálculo en ambos extremos, el tiempo total del in-

tercambio resulta menor (Costello et al., 2016). SIDH además utiliza compresión, por lo que es uno de los algoritmos que utiliza claves relativamente más pequeñas. Aunque no se conocen ataques que vulneren a SIDH, este tipo de problemas comenzó a estudiarse recientemente, y se necesita más tiempo de análisis para que pueda ganar confianza respecto de su seguridad post-cuántica.

Entre los algoritmos basados en SIDH se encuentra el mecanismo de encapsulamiento de claves SIKE (Jao et al., 2019). SIKE ofrece los niveles de seguridad recomendados 1, 2, 3 y 5 (véase Sección V.2). El principal inconveniente de SIKE es que su rendimiento no es mejor que el de otros esquemas de cifrado de curva elíptica clásicos, o esquemas post-cuánticos. Igualmente se ha realizado experimentos que mejoran el rendimiento de SIKE reduciendo el tiempo total de cálculo (Koziel et al., 2018, 2016).

A diferencia de otros competidores bien conocidos como NTRU y RingLWE, SIDH/SIKE soporta PFS, por lo que si un atacante obtuviera la clave de sesión actual, esto no comprometerá las sesiones futuras.

V.1.6 Criptografía de clave secreta

Como se mencionó en la Sección IV.4.2 muchos algoritmos de clave simétrica actuales se supone que son resistentes a ataques cuánticos con solamente incrementar el tamaño de la clave. El ejemplo que lidera en esta categoría es el cifrador Daemen-Dijmen “Rijndael”, conocido como AES.

V.2 PROCESO DE ESTANDARIZACIÓN

Los algoritmos de cifrado post-cuántico aún no se encuentran estandarizados, y es por ello que el el Instituto Nacional de Estándares y Tecnología de los Estados Unidos, NIST (NIST, 2020) promueve un concurso que tiene como objetivo evaluar y generar estándares de cifrado post-cuántico para la

industria. Para ello solicita a la comunidad criptográfica algoritmos de clave pública que se crean resistentes a ataques cuánticos con la intención de analizar alternativas (NIST, 2018).

Este concurso lanzó su primera ronda de recepción de propuestas en el 2017, la segunda en el 2019 y la tercera el 22 de julio de 2020 (NIST, 2018). El proceso de estandarización debería concluir entre el 2022 y el 2024 (NAP, 2019). Cada ronda finaliza con la presentación de algoritmos candidatos en las conferencias PQCrypto y Conferencia de estandarización de criptografía post-cuántica. Los algoritmos que se posiciones como candidatos serios podrán ser considerados por instituciones de estandarización como el Grupo de Trabajo de Ingeniería de Internet (IETF, por sus siglas en inglés), la Organización Internacional para la Estandarización) (ISO, por sus siglas en inglés) o la Unión Internacional de Telecomunicaciones (ITU, por sus siglas en inglés). Con estos estándares los proveedores de servicios en Internet podrán comenzar a incorporar criptografía post-cuántica en su infraestructura.

El NIST planteó una lista de niveles de seguridad a los que deben ajustarse las propuestas. Estos niveles representan la fortaleza del algoritmo a un ataque cuántico, y tienen un equivalente con algoritmos resistentes actuales. Se han definido cinco niveles para los nuevos algoritmos, siendo el nivel 1 el que comprende los algoritmos más débiles, y el 5 el que incluye a los más fuertes. La Tabla 1 muestra la equivalencia entre estos niveles y la resistencia cuántica de los algoritmos actuales considerados.

Tabla 1: Niveles de seguridad del NIST y equivalencias.

Nivel de Seguridad	Equivalencia	Resistencia Cuántica
L1	AES-128	Débil
L2	SHA-256/SHA3-256	Fuerte
L3	AES-192	Más fuerte
L4	SHA-384/SHA3-384	Muy fuerte
L5	AES-256	El más fuerte

Si bien el NIST tiene en cuenta que estas cinco categorías son resistentes a ataques cuánticos, hace algunas aclaraciones. Los algoritmos de nivel L1 se consideran probablemente seguros a menos que las computadoras cuánticas mejoren su rendimiento más rápido de lo que se prevé. Es por ello que muchos expertos no consideran a los algoritmos L1 verdaderamente seguros a largo plazo. El NIST igualmente los publica como alternativas aceptables al momento de liberar estándares, los califica como un paso intermedio a esquemas más seguros en el mediano plazo.

Los niveles L2 y L3 se consideran probablemente seguros de forma previsible, y los niveles L4 y L5 se consideran muy seguros. Los algoritmos L4 y L5 pueden ser difíciles de llevar a la práctica por su arquitectura e implementación, y además, en general, no tiene buen rendimiento.

La mayoría de los investigadores que presentan sus propuestas en general lo hacen para los niveles L1, L3 y L5, aunque hay excepciones (Prime, 2020; Hamburg, 2019; Ducas et al., 2018; Chen et al., 2020; Falcon, 2020).

Existen variantes híbridas que combinan algoritmos post-cuánticos con curvas elípticas NIST tradicionales según su nivel de seguridad. Los niveles de seguridad para cada algoritmo propuesto pueden verse en (OQS, 2021a).

VI IMPLEMENTACIONES

Este capítulo introduce las implementaciones de software que se analizarán en la parte DESARROLLO, y sobre las que se realizarán los experimentos.

VI.1 PROYECTO OPEN-QUANTUM-SAFE

Open Quantum Safe (OQS) (Stebila y Mosca, 2017) es un proyecto destinado al desarrollo y prototipado de soluciones basadas en criptografía post-cuántica. OQS consiste en dos lineas de desarrollo principales:

- liboqs: una biblioteca de código escrita en lenguaje C que provee algoritmos criptográficos resistentes a ataques cuánticos (OQS, 2021a).
- Una serie de prototipos que integran liboqs con protocolos y aplicaciones, incluida la ampliamente utilizada biblioteca OpenSSL.

El objetivo de este proyecto es asistir a las organizaciones en la migración de sus infraestructuras criptográficas a nuevos algoritmos resistentes a ataques cuánticos. Los administradores de sistemas y desarrolladores uso de mecanismos criptográficos para garantizar la seguridad de sus infraestructuras. Muchos de ellos se ven obligados a integrar algoritmos post-cuánticos, ya sea porque protegen información que requiere un nivel elevado de privacidad a largo plazo, o porque les preocupa que los algoritmos actuales sean eventualmente vulnerados. Con la intención de facilitar esta migración OQS provee prototipos de algoritmos post-cuánticos basados en código abierto para sus productos y aplicaciones. El diseño de OQS permite que los algoritmos post-cuánticos puedan ser adoptados sin mayores cambios en las mismas aplicaciones de software que antes utilizaban algoritmos tradicionales, tales como OpenSSL u OpenSSH.

Debe aclararse que, dado que los algoritmos post-cuánticos se encuentran en desarrollo experimental y no han sido evaluado exhaustivamente desarrolladores ni por la comunidad científica, se recomienda utilizar algoritmos híbridos, esquemas post-cuánticos combinados con otros clásicos.

VI.1.1 Liboqs y la arquitectura de OQS

OQS cuenta con la biblioteca liboqs en bajo o medio nivel, bajo licencia de código abierto y libre MIT, sobre la cual se montan una serie de integraciones de protocolos y aplicaciones de alto nivel. liboqs provee una interfaz común para los esquemas de intercambio de claves y firma digital, e implementa varios criptosistemas post-cuánticos. Algunas implementaciones están basadas en programas de código abierto existentes o son adaptaciones de éstos, y otras han sido escritas directamente para liboqs. Algunas de las funcionalidades provistas son las siguientes:

- Implementaciones de KEM (Grimes, 2019) basados en algoritmos post-cuánticos, y algoritmos de firma digital, ambos listados más abajo.
- Una Interfaz de Programación de Aplicaciones (API, por sus siglas en inglés) para poder utilizar estos algoritmos.
- Rutinas de verificación rendimiento.

liboqs también incluye algunas rutinas disponibles para todos sus módulos, entre las que se cuentan un generador de números aleatorios, primitivas de cifrado AES, funciones hash SHA-3, y scripts de prueba de rendimiento que permiten comparar las implementaciones soportadas.

El proyecto OQS además ofrece prototipos de integración de liboqs en protocolos de aplicación. La primer implementación se realizó sobre OpenSSL, una de las aplicaciones criptográfica más utilizadas. OpenSSL provee funciones de cifrado (libcrypto) y rutinas SSL/TLS (libssl) (véase Sección III).

OpenSSL brinda una capa de seguridad para protocolos de aplicación de TCP/IP como HTTPS y OpenVPN, objetivos de la presente tesis.

Si bien el proyecto OQS ofrece implementaciones de algoritmos de cifrado e intercambio de claves post cuánticos, éstos son experimentales al momento de escribir este trabajo, y se encuentran en proceso de estandarización.

VI.1.2 Encasulamiento de claves

Los algoritmos de encapsulamiento soportados por liboqs son (OQS, 2021a):

- BIKE: BIKE1-L1-CPA, BIKE1-L3-CPA, BIKE1-L1-FO, BIKE1-L3-FO
- Classic McEliece: **McEliece-348864, McEliece-348864f, McEliece-460896, McEliece-460896f, McEliece-6688128, McEliece-6688128f, McEliece-6960119, McEliece-6960119f, McEliece-8192128, McEliece-8192128f**
- FrodoKEM: FrodoKEM-640-AES, FrodoKEM-640-SHAKE, FrodoKEM-976-AES, FrodoKEM-976-SHAKE, FrodoKEM-1344-AES, FrodoKEM-1344-SHAKE
- Kyber: Kyber512, Kyber768, Kyber1024, Kyber512-90s, Kyber768-90s, Kyber1024-90s
- LEDAcrypt: LEDAcryptKEM-LT12, LEDAcryptKEM-LT32, **LEDAcryptKEM-LT52**
- NewHope: NewHope-512-CCA, NewHope-1024-CCA
- NTRU: NTRU-HPS-2048-509, NTRU-HPS-2048-677, NTRU-HPS-4096-821, NTRU-HRSS-701
- SABER: LightSaber-KEM, Saber-KEM, FireSaber-KEM
- SIKE: SIDH-p434, SIDH-p503, SIDH-p610, SIDH-p751, SIKE-p434, SIKE-p503, SIKE-p610, SIKE-p751, y sus versiones “compressed”
- ThreeBears: BabyBearEphem, BabyBear, MamaBearEphem, MamaBear, PapaBearEphem, PapaBear

Los algoritmos marcados en negrita hacen uso intensivo de pila, por lo que pueden fallar cuando ejecutan en multiproceso o en entornos multihilo.

VI.1.3 Esquemas de firma digital

Los esquemas de firma digital soportados por liboqs son (OQS, 2021a):

- CRYSTALS-Dilithium: Dilithium2, Dilithium3, Dilithium5 y sus variantes combinadas con AES.
- Falcon: Falcon-512, Falcon-1024
- MQDSS: MQDSS-31-48, MQDSS-31-64
- Picnic: versiones FS y UR de Picnic-L1, Picnic-L3, Picnic-L5, y Picnic2-L1-FS, Picnic2-L3-FS, Picnic2-L5-FS
- qTesla: qTesla-p-I, qTesla-p-III
- Rainbow: versiones Classic, Cyclic y Cyclic-Compressed de Rainbow-Ia, **Rainbow-IIIc**, **Rainbow-Vc**
- SPHINCS-<HASH>: versiones simple y robust de SPHINCS-<HASH>-128f, SPHINCS-<HASH>-128s, SPHINCS-<HASH>-192f, SPHINCS-<HASH>-192s, SPHINCS-<HASH>-256f, SPHINCS-<HASH>-256s para los hashes (<HASH>) Haraka, SHA256 y SHAKE256.

Al igual que el caso anterior, los algoritmos marcados en negrita pueden fallar al ejecutarse en entornos multiproceso o multihilo.

VI.1.4 Limitaciones de seguridad de liboqs

Según se vayan realizando avances en la investigación y el desarrollo de liboqs los algoritmos soportados pueden cambiar y llegar a ser considerados inseguros comparados con los algoritmos tradicionales.

Debido a que el proceso de estandarización del NIST es más lento (véase Sección V.2), que la implementación de algoritmos tentativos en lenguajes

de programación, puede ocurrir que liboqs provea algoritmos que todavía no están aprobados ni estandarizados, y por tanto pueden contener fallos y vulnerabilidades. Es recomendable, por lo tanto, utilizar alternativas híbridas en las que los algoritmos de clave pública post-cuánticos se combinen con otros asimétricos tradicionales como RSA o variantes de curva elíptica.

VI.2 PROYECTO WOLFSSL

wolfSSL, anteriormente CyaSSL (WolfSSL, 2020c), es una biblioteca de cifrado SSL/TLS de código abierto, ligera, portable y escrita en lenguaje C. Su principal objetivo es proveer una implementación de protocolos SSL/TLS ligera pensada para entornos de bajos recursos de hardware. Debido su reducido tamaño y buen rendimiento, wolfSSL es ideal para dispositivos de Internet de las Cosas (IoT, por sus siglas en inglés), software embebido y Sistemas Operativos de Tiempo Real (RTOS, por sus siglas en inglés).

wolfSLL puede correr tanto sobre computadoras de escritorio como en entornos empresariales y en computación de nube. Soporta los estándares de la industria TLS v1.3 y DTLS v1.2, y su tamaño es hasta 20 veces menor que el de alternativas como OpenSSL. wolfSSL ofrece una API y una capa de compatibilidad con OpenSSL, y soporta el protocolo de comprobación de certificados digitales (OCSP, por sus siglas en inglés) (Myers et al., 1999) y la lista de renovación de certificados (CRL, por sus siglas en inglés) (Cooper et al., 2008).

Como backend criptográfico utiliza la biblioteca wolfCrypt, un motor criptográfico ligero escrito en ANSI C pensado para su uso en sistemas con recursos limitados. wolfCrypt soporta algoritmos de cifrado comunes como RSA, ECC, DSS, DH/ECDH, NTRU, DES, 3DES, AES, Camellia, IDEA, ARC4, HC-128, ChaCha20, MD2, MD4, MD5, SHA-1, SHA-2, SHA-3, BLAKE2, RIPEMD-160 y Poly1305, y también soporta algoritmos experimentales co-

mo RABBIT, un software de transmisión de códigos de dominio público publicado por el Proyecto eSTREAM de la Unión Europea (ECRYPT, 2008), útil para cifrar datos en entornos de alto rendimiento. wolfSSL también soporta la curva elíptica Curve25519 y su esquema de firma, Ed25519.

Al momento de realizar esta tesis wolfSSL incluía el algoritmo de criptografía asimétrica post-cuántica NTRU. Como se comentará en la Sección IX, este algoritmo no está presente en versiones actuales de wolfSSL.

VI.3 TRABAJOS RELACIONADOS

En esta sección mencionan algunos trabajos relacionados a la temática de esta tesis, y que pueden servir para complementar o ampliar los experimentos descritos en la Sección DESARROLLO. La mayoría de las publicaciones académicas relacionadas hacen referencia a la criptografía post-cuántica subyacente a las implementaciones de software analizadas, por lo que exceden el alcance de este trabajo. Se mencionan a continuación aquellas publicaciones, académicas y no académicas, que se refieren a implementaciones de software y librerías utilizadas como base para los experimentos.

(IBM, 2022) realiza, de manera automatizada, algunos experimentos similares a los planteados en la presente tesis. Se trata de Key Protect (IBM, 2021), una solución de software de IBM que permite almacenar datos en forma segura en su servicio IBM Cloud. Esta implementación recientemente añadió criptografía post-cuántica en TLS utilizando el algoritmo Kyber (Ducas et al., 2019) sobre plataformas Linux. Key Project dispone de dos modos de trabajo:

- Modo Seguro Cuántico: este modo utiliza sólo el algoritmo kyber<S>, donde <S> representa el tamaño de la clave (512, 768 y 1024).
- Modo Híbrido: este modo utiliza una combinación del algoritmo post-cuántico con algoritmos clásicos. Aquí los algoritmos híbridos sopor-

tados son p<S>_kyber<S>, donde <S> es el tamaño de la clave (512, 768 y 1024), y combina kyber con ECDH usando la curva NIST p<S>.

El modo híbrido se basa en las recomendaciones del proyecto OQS, y en el sitio oficial del Key Protect de IBM se reseñan los pasos para la compilación e instalación automatizada utilizando, liboqs y OQS-OpenSSL v1.1.1.

Por otro lado, (Parkhomenko, 2021) realiza una introducción a la criptografía post-cuántica y analiza el impacto de los ataques cuánticos sobre los algoritmos criptográficos actuales. Como implementación de algoritmos resistentes a ataques cuánticos menciona al proyecto OQS, a su librería liboqs, y su integración con OpenVPN en PQCrypto-VPN, analizado en Sección VIII.7 de esta tesis. El artículo explica los pasos básicos para montar un entorno de pruebas y con él generar claves y certificados digitales.

wolfSSL también realizó recientemente una integración con liboqs implementando criptografía post-cuántica híbrida (WolfSSL, Inc., 2021a). El desarrollo se basó en el diseño propuesto por (Steblia et al., 2021), y en la última versión del Manual de wolfSSL (WolfSSL, 2021) pueden encontrarse los pasos necesarios para integrar wolfSSL con liboqs. Además, recientemente añadió soporte para el esquema de firma digital FALCON (Falcon, 2020) y puede ser incorporado a dicha integración (WolfSSL, Inc., 2022b).

Otro trabajo relacionado en parte con esta tesis es (George et al., 2021), que analiza el rendimiento de TLS 1.3 integrando las implementaciones de los algoritmos Kyber, Saber, Dilithium y Falcon de la librería PQM4 (Kannwischer et al., 2022) en wolfSSL. La intención del trabajo es analizar el rendimiento de los mencionados algoritmos en dispositivos embebidos basados en procesadores ARM Cortex-M4.

Finalmente (Bernstein et al., 2021) analiza mejoras de rendimiento en la negociación TLS 1.3 usando NTRU con OpenSSL, en lo que denomina OpenSSLNTRU. Esta implementación representa una alternativa a la inte-

gración OQS-OpenSSL analizada en esta tesis. Como ventaja sobre OQS-OpenSSL, OpenSSLNTRU requiere parches más pequeños sobre OpenSSL, lo que facilita su integración. Como desventaja, OpenSSLNTRU solamente soporta un candidato para estandarización del NIST, NTRU, mientras que OQS-OpenSSL soporta casi todos los algoritmos candidatos.

En la Sección X se mencionan otras implementaciones de software y/o de librerías criptográficas que incorporan criptografía post-cuántica tanto en TLS como en otros protocolos de red, y que pueden servir para dimensionar la penetración de estos nuevos algoritmos.

DESARROLLO

En esta sección se realiza un análisis minucioso de la integración de la biblioteca liboqs provista por el proyecto OQS, con dos versiones de OpenSSL, v1.0.2 y v1.1.1, y las experimentaciones realizadas con dichas impementaciones. Un análisis similar se lleva a cabo con wolfSSL y su integración con NTRU. Finalmente se reseñan otras implementaciones no consideradas en el presente trabajo, algunas presentes al iniciar el desarrollo de esta tesis, y otras surgidas con posterioridad.

VII ANÁLISIS DE OQS-OPENSSL

El proyecto OQS integró la biblioteca liboqs con un fork de OpenSSL la que denominó OQS-OpenSSL, para proveer un prototipo que soporte intercambio de claves, autenticación y cipher suites post-cuánticos. El objetivo de esta integración es brindar un prototipado sencillo y experimental de criptografía post-cuántica, y no debería ser considerado una suite de seguridad estable para su uso en entornos de producción. OQS realizó la integración de liboqs en dos versiones de OpenSSL: 1.0.2 y 1.1.1. En el presente trabajo se analizaron ambas versiones, sus algoritmos soportados y consideraciones de seguridad. Se estudió su integración con OpenVPN, Apache y Nginx. En todos los casos se utilizó la útlima versión estable de liboqs al momento de realizar las pruebas, la v0.3.0.

Nota de obsolescencia

OpenSSL suspendió el soporte para la versión 1.0.2 el 1 de Enero de 2020, como resultado, el proyecto OQS decidió discontinuar el soporte de esta versión de OQS-OpenSSL, por lo que ya no se actualizará su repositorio Git. Debido a esto los nuevos proyectos que requieran implementar criptografía post-cuántica en TLS deberían utilizar OQS-OpenSSL v1.1.1. Igualmente, dado que la versión v1.1.1 de OQS-OpenSSL no soporta negociación de cipher suites post-cuánticas en TLS 1.3, el presente trabajo estudió la versión 1.0.2 con la intención de analizar dicha negociación en TLS v1.2.

VII.1 ANÁLISIS DE OQS-OPENSSL v1.0.2

OQS-OpenSSL 1.0.2 es un fork de OpenSSL v1.0.2 integrado con la librería liboqs que facilita el armado de prototipos de criptografía post-cuántica en TLS v1.2. Este fork soporta los siguientes mecanismos criptográficos:

- Intercambio de claves post-cuántico.
- Intercambio de claves híbrido (post-cuántico y curva elíptica).
- Primitivas post-cuánticas de liboqs en el comando speed de OpenSSL

OQS-OpenSSL v1.0.2 sólo permite comunicaciones en TLS1.2 y brinda mecanismos de intercambio de claves post-cuánticos e híbridos, a los que añade de negociación de cipher suites resistentes a ataques cuánticos. En el presente trabajo se realizaron pruebas de concepto del intercambio de claves híbrido y de negociación de cipher suites post-cuánticas en TLS.

OQS-OpenSSL v1.0.2 no debería ser utilizado en producción debido a que su línea de desarrollo es experimental y, además, esta versión se considera obsoleta, tal y como se mencionó anteriormente. Este proyecto desarrolla un prototipo de criptografía post-cuántica previo a la estandarización de nuevos algoritmos por parte del NIST, por lo que es también recomendable remitirse a criptografía híbrida para montar servicios que sean, como mínimo, tan seguros como los algoritmos criptográficos tradicionales.

VII.1.1 Cipher suites disponibles

Los algoritmos de intercambio de claves y cipher suites disponibles en esta versión de OQS-OpenSSL pueden listarse utilizando el comando `ciphers` del OpenSSL compilado. La Sección VIII.2 amplía estos detalles y analiza capturas de tráfico de red. En general estas cipher suites post-cuánticas incluyen el prefijo `OQSKEM-DEFAULT` en su definición. OQS-OpenSSL v1.0.2 soporta cipher suites tradicionales y post-cuánticas.

VII.1.2 Algoritmos de Intercambio de claves

Los algoritmos de intercambio de claves soportados por este fork son:

- `DEFAULT`
- `DEFAULT-ECDHE`: `DEFAULT` en modo híbrido con ECDH.

`DEFAULT` indica que, durante la negociación KEM, se haga uso del algoritmo configurado por defecto durante la compilación de liboqs. `DEFAULT-ECDHE` es el mecanismo KEX predeterminado en modo híbrido con ECDHE. El Alg. 3 muestra un fragmento del archivo `src/kem/kem.c` de código fuente de liboqs. En dicho fragmento se ve el modo en el que se selecciona del algoritmo KEM predeterminado. La función `OQS_KEM_alg_is_enabled` retornará `OQS_KEM_DEFAULT` si el método de intercambio de claves por defecto es `OQS_KEM_alg_default`. En caso contrario se verificará, contra una lista de algoritmos válidos, el algoritmo que fue pasado por argumento, `method_name`, y retornará 1 si dicho algoritmo se encuentra habilitado.

```
36 OQS_API int OQS_KEM_alg_is_enabled(const char *method_name
       ) {
37   if (method_name == NULL) {
38     return 0;
39   }
40   if (0 == strcasecmp(method_name, OQS_KEM_alg_default)) {
41     return OQS_KEM_alg_is_enabled(OQS_KEM_DEFAULT);
42
43   } else if (0 == strcasecmp(method_name,
       OQS_KEM_alg_bike1_l1)) {
44 #ifdef OQS_ENABLE_KEM_bike1_l1
45     return 1;
46 #else
47     return 0;
48 #endif
```

Algoritmo 3: OQS-OpenSSLv1.0.2. Selección de algoritmo KEM.

La lógica de selección del algoritmo KEM predeterminado se encuentra en el script de configuración de liboqs que debe ejecutarse previo a la compilación. Esto puede apreciarse en el Alg. 4. La variable `$KEM_DEFAULT` recibirá, de manera predeterminada, el algoritmo `OQS_KEM_alg_sike_p503`, salvo que se configure otro diferente durante la compilación.

```
21634 case "$KEM_DEFAULT" in
21635 #(
21636 "")
21637
21638 $as_echo "#define OQS_KEM_DEFAULT OQS_KEM_alg_sike_p503"
        >>confdefs.h
21639
21640 ;;
21641 #(
21642 *)
21643
21644 cat >>confdefs.h <<_ACEOF
21645 #define OQS_KEM_DEFAULT $KEM_DEFAULT
21646 _ACEOF
21647
21648 ;;
21649 esac
```

Algoritmo 4: OQS-OpenSSLv1.0.2. Configuración de algoritmo KEM.

Una vez que el algoritmo fue seleccionado, la instalación de liboqs cargará dicho algoritmo en la cabecera `openssl/oqs/include/oqs/oqsconfig.h`, utilizada por OQS-OpenSSL. El Alg. 5 muestra un fragmento de este archivo, la línea 93 especifica el algoritmo predeterminado en este caso.

```
93 #define OQS_KEM_DEFAULT OQS_KEM_alg_sike_p503
```

Algoritmo 5: OQS-OpenSSLv1.0.2. Algoritmo KEM predeterminado.

Si se quisiera utilizar otro algoritmo KEM se deberá añadir un parámetro adicional en la llamada a `./configure` especificando el contenido de la variable `OQS_KEM_DEFAULT`. Por ejemplo, para configurar `FrodoKEM-640-AES` debe utilizarse la siguiente macro en la llamada a `./configure`:

```
    -DOQS_KEM_DEFAULT="OQS_KEM_alg_frodokem_640_aes"
```

Los valores de macros válidos se encuentran definidos en el archivo de cabecera `src/kem/kem.h`, y puede ser extendida para añadir nuevas implementaciones. El Alg. 6 muestra un fragmento de este archivo, en el que se ven las variantes del algoritmo FrodoKEM.

```
78 /** Algorithm identifier for FrodoKEM-640-AES KEM. */
79 #define OQS_KEM_alg_frodokem_640_aes "FrodoKEM-640-AES"
80 /** Algorithm identifier for FrodoKEM-640-SHAKE KEM. */
81 #define OQS_KEM_alg_frodokem_640_shake "FrodoKEM-640-SHAKE
    "
82 /** Algorithm identifier for FrodoKEM-976-AES KEM. */
83 #define OQS_KEM_alg_frodokem_976_aes "FrodoKEM-976-AES"
84 /** Algorithm identifier for FrodoKEM-976-SHAKE KEM. */
85 #define OQS_KEM_alg_frodokem_976_shake "FrodoKEM-976-SHAKE
    "
86 /** Algorithm identifier for FrodoKEM-1344-AES KEM. */
87 #define OQS_KEM_alg_frodokem_1344_aes "FrodoKEM-1344-AES"
88 /** Algorithm identifier for FrodoKEM-1344-SHAKE KEM. */
89 #define OQS_KEM_alg_frodokem_1344_shake "FrodoKEM-1344-
    SHAKE"
```

Algoritmo 6: OQS-OpenSSLv1.0.2. Definición de macros para KEM.

La siguiente lista muestra las macros válidas en este caso.

- `OQS_KEM_alg_default`
- `OQS_KEM_alg_<B>_<L>`, donde `<B>` puede ser `bike1`, `bike2` o `bike3`, y `<L>` puede ser L1, L3 o L5.
- `OQS_KEM_alg_kyber_<S>`, donde `<S>` puede ser 512, 768 o 1024.
- `OQS_KEM_alg_newhope_512cca`, `OQS_KEM_alg_newhope_1024cca`.
- `OQS_KEM_alg_ntru_<V>`, donde `<V>` puede ser `hps2048509`, `hps2048677`, `hps4096821` o `hrss701`.
- `OQS_KEM_alg_saber_<V>`, donde `<V>` puede ser `lightsaber`, `saber` o `firesaber`.
- `OQS_KEM_alg_frodokem_<L>_<C>`, donde `<L>` puede ser 640, 976 o 1344, y `<C>` puede ser `aes` o `shake`.
- `OQS_KEM_alg_<V>_<L>`, donde `<V>` puede ser `sidh` o `sike`, y `<L>` puede ser `p434`, `p503`, `p610` o `p751`.

Una vez elegido el mecanismo KEM y compilados librería liboqs y OQS-OpenSSL, la suite dispondrá de las siguientes cipher suites, donde **<KEX>** puede ser `DEFAULT` o `DEFAULT-ECDHE`.

- OQSKEM-**<KEX>**-RSA-AES128-GCM-SHA256

- OQSKEM-**<KEX>**-ECDSA-AES128-GCM-SHA256
- OQSKEM-**<KEX>**-RSA-AES256-GCM-SHA384
- OQSKEM-**<KEX>**-ECDSA-AES256-GCM-SHA384

VII.1.3 Algoritmos de Firma Digital

El algoritmo de firma digital utilizado por defecto es Dilithium2. Al igual que en el caso anterior, puede configurarse este algoritmo modificando el código fuente de la implementación al configurar liboqs. El Alg. 7 muestra un fragmento del código fuente `src/sig/sig.h` de liboqs, donde se ve la lógica de decisión para la selección del algoritmo de firma predeterminado. De manera similar a como ocurría con los algoritmos KEM, la función `OQS_SIG_alg_is_enabled` retornará `OQS_SIG_DEFAULT` si el método de firma por defecto es `OQS_SIG_alg_default`. En caso contrario se verificará, contra una lista de algoritmos válidos, el algoritmo que fue pasado por argumento, `method_name`, y retornará 1 si dicho algoritmo se encuentra habilitado.

```
33 OQS_API int OQS_SIG_alg_is_enabled(const char *method_name
     ) {
34   if (method_name == NULL) {
35     return 0;
36   }
37   if (0 == strcasecmp(method_name, OQS_SIG_alg_default)) {
38     return OQS_SIG_alg_is_enabled(OQS_SIG_DEFAULT);
39     /////
      OQS_COPY_FROM_PQCLEAN_FRAGMENT_ENABLED_CASE_START
40   } else if (0 == strcasecmp(method_name,
      OQS_SIG_alg_dilithium_2)) {
41 #ifdef OQS_ENABLE_SIG_dilithium_2
42     return 1;
43 #else
44     return 0;
45 #endif
46 [...]
```

Algoritmo 7: OQS-OpenSSLv1.0.2. Selección de algoritmo de firma.

El Alg. 8 muestra las líneas del archivo `liboqs/configure` que cargan el algoritmo previo a la compilación. La variable `$SIG_DEFAULT` recibirá, de manera predeterminada, el algoritmo de firma `OQS_SIG_alg_dilithium_2`, sal-

vo que se configure alguna alternativa antes de compilar de liboqs. Si se quisiera cambiar el algoritmo predeterminado debería especificarse un valor para la macro `OQS_SIG_DEFAULT` durante la llamada a `./configure`.

```
21652 case "$SIG_DEFAULT" in
21653 #(
21654 "")
21655
21656   $as_echo "#define OQS_SIG_DEFAULT
        OQS_SIG_alg_dilithium_2" >>confdefs.h
21657
21658 ;;
21659 #(
21660 *)
21661
21662   cat >>confdefs.h <<_ACEOF
21663   #define OQS_SIG_DEFAULT $SIG_DEFAULT
21664   _ACEOF
21665
21666 ;;
21667 esac
```

Algoritmo 8: OQS-OpenSSLv1.0.2. Configuración de algoritmo de firma.

El script `./configure` cargará el algoritmo especificado en `OQS_SIG_DEFAULT` dentro del archivo `openssl/oqs/include/oqs/oqsconfig.h`. El Alg. 9 muestra un fragmento de dicho archivo, donde puede verse el algoritmo utilizado de manera predeterminada.

```
95 #define OQS_SIG_DEFAULT OQS_SIG_alg_dilithium_2
```

Algoritmo 9: OQS-OpenSSLv1.0.2. Algoritmo SIG predeterminado.

Para cambiar el algoritmo predeterminado puede modificarse el valor de la macro `OQS_SIG_DEFAULT` en la llamada a `./configure`. Por ejemplo, a continuación se muestra la definición de macro que configura qTESLA_I como algoritmo de firma:

```
    -DOQS_SIG_DEFAULT="OQS_SIG_alg_qTESLA_I"
```

Los valores de macros válidos están almacenados en el archivo de cabecera `src/sig/sig.h`. El Alg. 10 muestra un fragmento del mismo en el que se definen las tres versiones de qTESLA soportadas.

```
52 /** Algorithm identifier for qTESLA_I */
53 #define OQS_SIG_alg_qTESLA_I "qTESLA_I"
54 /** Algorithm identifier for qTESLA_III_size */
55 #define OQS_SIG_alg_qTESLA_III_size "qTESLA_III_size"
56 /** Algorithm identifier for qTESLA_III_speed */
57 #define OQS_SIG_alg_qTESLA_III_speed "qTESLA_III_speed"
```

Algoritmo 10: OQS-OpenSSLv1.0.2. Macros de firma digital.

Los nombres de macros válidas se listan a continuación.

- `OQS_SIG_alg_default`
- `OQS_SIG_alg_picnic_<L>_<S>`

  Donde `<L>` puede ser `L1`, `L3` o `L5`, y `<S>` puede ser `FS` o `UR`.
- `OQS_SIG_alg_picnic2_L1_FS`, `OQS_SIG_alg_picnic2_L3_FS`, `OQS_SIG_alg_picnic2_L5_FS`
- `OQS_SIG_alg_qTESLA_I`, `OQS_SIG_alg_qTESLA_III_size`, `OQS_SIG_alg_qTESLA_III_speed`
- `OQS_SIG_alg_dilithium_2`, `OQS_SIG_alg_dilithium_3`, `OQS_SIG_alg_dilithium_4`
- `OQS_SIG_alg_mqdss_31_48`, `OQS_SIG_alg_mqdss_31_64`
- `OQS_SIG_alg_sphincs_<H>_<L>_<S>`

  `<L>` puede ser 128f, 128s, 192f, 192s, 256f y 256s,

  y `<S>` puede ser `robust` o `simple`.

VII.2 ANÁLISIS DE OQS-OPENSSL v1.1.1

Esta rama de OQS-OpenSSL es un fork de OpenSSL v1.1.1 que integra liboqs para incluir algoritmos de intercambio de claves y de autenticación post-cuánticos. Al igual que la v1.0.2 comentada en la Sección VII.1, esta versión se encuentra en una etapa de desarrollo experimental, por lo que no debería utilizadarse en entornos de producción, ni se debe confiar en sus mecanismos de protección. Esta rama incluye algoritmos de intercambio de claves post-cuánticos en TLS v1.3 (a diferencia de la v1.0.2 que lo hacía

en TLS v1.2). Además la v1.1.1 añade algoritmos asimétricos resistentes a ataques cuánticos, por lo que puede generar claves públicas y privadas para cifrado y autenticación post-cuántica. Es un desarrollo muy activo al día de la fecha, y hasta el momento tiene las siguientes características:

- Intercambio de claves post-cuántico.
- Intercambio de claves híbrido (post-cuántico y curva elíptica).
- Autenticación post-cuántica.
- Autenticación híbrida (post-cuántica y curva elíptica).
- Soporte para Sintaxis de Mensaje Criptográfico (CMS, por sus siglas en inglés) (Housley, 2009), firma y verificación usando cualquiera de los algoritmos post-cuánticos soportados.

Esta versión se encuentra en etapas iniciales de desarrollo y no provee soporte para cipher suites post-cuánticas en TLS v1.3. Además la integración con liboqs es experimental, por lo que es recomendable hacer uso de mecanismos híbridos que incorporen algoritmos tradicionales.

VII.2.1 Algoritmos de encapsulamiento de claves

Los algoritmos de encapsulamiento de claves soportados por esta versión de OQS-OpenSSL pueden verse en la siguiente lista (OQS, 2021b):

- BIKE: bike1l1, bike1l3
- FrodoKEM: frodo640aes, frodo640shake, frodo976aes, frodo976shake, frodo1344aes, frodo1344shake
- CRYSTALS-Kyber: kyber512, kyber768, kyber1024, kyber90s512, kyber90s768, kyber90s1024
- HQC: hqc128, hqc192, **hqc256**
- NTRU: ntru_hps2048509, ntru_hps2048677, ntru_hps4096821, ntru_hrss701, ntru_hps40961229, ntru_hrss1373

- NTRU-Prime: ntrulpr653, ntrulpr761, ntrulpr857, ntrulpr1277, sntrup653, sntrup761, sntrup857, sntrup1277
- SABER: lightsaber, saber, firesaber
- SIDH/SIKE: sidhp434, sidhp503, sidhp610, sidhp751,sikep434, sikep503, sikep610, sikep751

El algoritmo marcado en negrita hace uso intensivo de la pila de proceso, por lo que puede ocasionar fallas cuando se implementa en multiproceso o multihilo. El algoritmo `oqs_kem_default` permite indicar a OQS-OpenSSL que haga uso del mecanismo de intercambio de claves que se estableció de manera predeterminada durante la configuración de liboqs. Es equivalente al mecanismo `DEFAULT` de OQS-OpenSSL v1.0.2 visto en la sección VII.1.2.

El proyecto brinda algoritmos post-cuánticos puros y variantes híbridas que combinan los esquemas listados anteriormente con ECDH. Las curvas utilizadas en este caso dependen del nivel de seguridad de uno cada algoritmo. Así, si se supone que `<KEX>` es uno de los algoritmos listados, esta versión de OQS-OpenSSL provee los siguientes mecanismos híbridos:

- Si `<KEX>` tiene un nivel de seguridad L1, este fork provee el algoritmo híbrido `p256_<KEX>`, que combina `<KEX>` y ECDH con la curva P256.
- Si `<KEX>` tiene un nivel de seguridad L3, este fork provee el algoritmo híbrido `p384_<KEX>`, que combina `<KEX>` y ECDH con la curva P384.
- Si `<KEX>` tiene un nivel de seguridad L5, este fork provee el algoritmo híbrido `p521_<KEX>`, que combina `<KEX>` y ECDH con la curva P521.

Por ejemplo, la suite soporta el algoritmo post-cuántico puro Kyber768, que tiene un nivel de seguridad L3, por lo que también se dispone del algoritmo híbrido p384_kyber768, que combina a Kyber768 con ECDH curva P384.

Inicialmente, al igual que en la versión 1.0.2 de OQS-OpenSSL, la selección del algoritmo KEM se realizaba en el archivo fuente `src/kem/kem.c`

del código de liboqs. Sin embargo hoy se considera una macro desactualizada. La biblioteca liboqs en este caso activa de manera predeterminada todos los algoritmos listados en los párrafos anteriores, y con ello habilita a OpenSSL a seleccionar, mediante configuraciones, el algoritmo a utilizar durante la negociación TLS. Para cambiar estas configuraciones iniciales ya no se utiliza el script `configure`, en su lugar las opciones de configuración se pasan a la llamada `cmake`. De manera predeterminada todos los algoritmos mencionados estarán habilitados, pero pueden desactivarse selectivamente definiendo macros de configuración durante la llamada a `cmake`. Esta configuración se verá en la Sección VIII.4.

VII.2.2 Algoritmos de autenticación y firma digital

OQS-OpenSSL v1.1.1 también soporta los siguientes mecanismos de autenticación y firma digital, asumiendo que fueron compilados previamente en liboqs. La lista completa puede verse a continuación (OQS, 2021b).

- CRYSTALS-Dilithium: dilithium2, dilithium3, dilithium5, dilithium2_aes, dilithium3_aes, dilithium5_aes
- Falcon: falcon512, falcon1024
- Picnic: picnicl1fs, picnicl1ur, picnicl1full, picnic3l1, picnic3l3, picnic3l5
- Rainbow: rainbowI<V>, rainbowIII<V>, rainbowV<V>
  Donde <V> puede ser *classic*, *circumzenithal* o *compressed*.
- SPHINCS: sphincs<H><L><S>
  Donde <H> puede ser *haraka*, *sha256* o *shake256*, <L> puede ser 128f, 128s, 192f, 192s, 256f y 256s, y <S> puede ser *robust* o *simple*.

Esta versión también soporta variantes híbridas que combinan los algoritmos listados arriba con otros tradicionales. Suponiendo que <SIG> sea uno de los algoritmos post-cuánticos listados:

- Si <SIG> tiene un nivel de seguridad L1 o L2, OQS-OpenSSL provee métodos rsa3072_<SIG> y p256_<SIG> que combinan <SIG> con RSA3072 y con ECDSA con curva NIST P256 respectivamente.
- Si <SIG> tiene un nivel de seguridad L3 o L4, OQS-OpenSSL provee el método p384_<SIG> que combina <SIG> con ECDSA con curva P384.
- Si <SIG> tiene un nivel de seguridad L5, OQS-OpenSSL provee el método p521_<SIG> que combina <SIG> con ECDSA con curva P521.

Por ejemplo, Dihithium2 tiene un nivel de seguridad L2, por lo que esta versión de OQS-OpenSSL ofrece de los algoritmos híbridos rsa3072_dilithium2 y p256_dilithium2. La lista completa de los algoritmos soportados y su nivel de seguridad puede leerse en la documentación de (OQS, 2021a).

Los algoritmos de firma digital mencionados se habilitan de manera predeterminada en liboqs (y posteriormente en OpenSSL). Si se desea desactivar algún algoritmo puede hacerse redefiniendo una macro de configuración durante la llamada a cmake. Esto se comentará en la Sección VIII.4.

Estos algoritmos se encuentran definidos dentro del archivo de código fuente openssl/ssl/ssl_local.h. El Alg. 11 muestra un fragmento de dicho archivo, en el que se ve las macros que definen cada uno de los algoritmos de clave pública, particularmente los post-cuánticos y los híbridos, de especial interés para este trabajo.

El mismo archivo de código fuente define los algoritmos de firma digital, y los correspondientes códigos que se intercambian por la red cuando se realiza la negociación TLS. El Alg. 12 muestra un fragmento de dicho código fuente donde se aprecian las macros de definición de estos algoritmos. Estos códigos serán de utilidad para entender los experimentos llevados a cabo en la Sección VIII.4.

```
468 #define SSL_PKEY_DILITHIUM2 9
469 #define SSL_PKEY_P256_DILITHIUM2 10
470 #define SSL_PKEY_RSA3072_DILITHIUM2 11
471 #define SSL_PKEY_DILITHIUM3 12
472 #define SSL_PKEY_P384_DILITHIUM3 13
473 #define SSL_PKEY_DILITHIUM5 14
474 #define SSL_PKEY_P521_DILITHIUM5 15
475 #define SSL_PKEY_DILITHIUM2_AES 16
476 #define SSL_PKEY_P256_DILITHIUM2_AES 17
477 #define SSL_PKEY_RSA3072_DILITHIUM2_AES 18
478 #define SSL_PKEY_DILITHIUM3_AES 19
479 #define SSL_PKEY_P384_DILITHIUM3_AES 20
480 #define SSL_PKEY_DILITHIUM5_AES 21
481 #define SSL_PKEY_P521_DILITHIUM5_AES 22
482 #define SSL_PKEY_FALCON512 23
483 #define SSL_PKEY_P256_FALCON512 24
484 #define SSL_PKEY_RSA3072_FALCON512 25
485 #define SSL_PKEY_FALCON1024 26
486 #define SSL_PKEY_P521_FALCON1024 27
487 #define SSL_PKEY_PICNICL1FULL 28
488 #define SSL_PKEY_P256_PICNICL1FULL 29
489 #define SSL_PKEY_RSA3072_PICNICL1FULL 30
490 #define SSL_PKEY_PICNIC3L1 31
491 #define SSL_PKEY_P256_PICNIC3L1 32
492 #define SSL_PKEY_RSA3072_PICNIC3L1 33
493 #define SSL_PKEY_RAINBOWICLASSIC 34
494 #define SSL_PKEY_P256_RAINBOWICLASSIC 35
495 #define SSL_PKEY_RSA3072_RAINBOWICLASSIC 36
496 #define SSL_PKEY_RAINBOWVCLASSIC 37
497 #define SSL_PKEY_P521_RAINBOWVCLASSIC 38
498 #define SSL_PKEY_SPHINCSHARAKA128FROBUST 39
499 #define SSL_PKEY_P256_SPHINCSHARAKA128FROBUST 40
500 #define SSL_PKEY_RSA3072_SPHINCSHARAKA128FROBUST 41
501 #define SSL_PKEY_SPHINCSSHA256128FROBUST 42
502 #define SSL_PKEY_P256_SPHINCSSHA256128FROBUST 43
503 #define SSL_PKEY_RSA3072_SPHINCSSHA256128FROBUST 44
504 #define SSL_PKEY_SPHINCSSHAKE256128FROBUST 45
505 #define SSL_PKEY_P256_SPHINCSSHAKE256128FROBUST 46
506 #define SSL_PKEY_RSA3072_SPHINCSSHAKE256128FROBUST 47
```

Algoritmo 11: OQS-OpenSSL v1.1.1. Algoritmos asimétricos post-cuánticos.

```
2487 ///// OQS_TEMPLATE_FRAGMENT_DEFINE_SIG_CODE_POINTS_START
2488 /* The following are all private use code points */
2489 #define TLSEXT_SIGALG_dilithium2 0xfea0
2490 #define TLSEXT_SIGALG_p256_dilithium2 0xfea1
2491 #define TLSEXT_SIGALG_rsa3072_dilithium2 0xfea2
2492 #define TLSEXT_SIGALG_dilithium3 0xfea3
2493 #define TLSEXT_SIGALG_p384_dilithium3 0xfea4
2494 #define TLSEXT_SIGALG_dilithium5 0xfea5
2495 #define TLSEXT_SIGALG_p521_dilithium5 0xfea6
2496 #define TLSEXT_SIGALG_dilithium2_aes 0xfea7
2497 #define TLSEXT_SIGALG_p256_dilithium2_aes 0xfea8
2498 #define TLSEXT_SIGALG_rsa3072_dilithium2_aes 0xfea9
2499 #define TLSEXT_SIGALG_dilithium3_aes 0xfeaa
2500 #define TLSEXT_SIGALG_p384_dilithium3_aes 0xfeab
2501 #define TLSEXT_SIGALG_dilithium5_aes 0xfeac
2502 #define TLSEXT_SIGALG_p521_dilithium5_aes 0xfead
2503 #define TLSEXT_SIGALG_falcon512 0xfe0b
2504 #define TLSEXT_SIGALG_p256_falcon512 0xfe0c
2505 #define TLSEXT_SIGALG_rsa3072_falcon512 0xfe0d
2506 #define TLSEXT_SIGALG_falcon1024 0xfe0e
2507 #define TLSEXT_SIGALG_p521_falcon1024 0xfe0f
2508 #define TLSEXT_SIGALG_picnicl1full 0xfe96
2509 #define TLSEXT_SIGALG_p256_picnicl1full 0xfe97
2510 #define TLSEXT_SIGALG_rsa3072_picnicl1full 0xfe98
2511 #define TLSEXT_SIGALG_picnic3l1 0xfe1b
2512 #define TLSEXT_SIGALG_p256_picnic3l1 0xfe1c
2513 #define TLSEXT_SIGALG_rsa3072_picnic3l1 0xfe1d
2514 #define TLSEXT_SIGALG_rainbowIclassic 0xfe27
2515 #define TLSEXT_SIGALG_p256_rainbowIclassic 0xfe28
2516 #define TLSEXT_SIGALG_rsa3072_rainbowIclassic 0xfe29
2517 #define TLSEXT_SIGALG_rainbowVclassic 0xfe3c
2518 #define TLSEXT_SIGALG_p521_rainbowVclassic 0xfe3d
2519 #define TLSEXT_SIGALG_sphincsharaka128frobust 0xfe42
2520 #define TLSEXT_SIGALG_p256_sphincsharaka128frobust 0xfe43
2521 #define TLSEXT_SIGALG_rsa3072_sphincsharaka128frobust 0
      xfe44
2522 #define TLSEXT_SIGALG_sphincssha256128frobust 0xfe5e
2523 #define TLSEXT_SIGALG_p256_sphincssha256128frobust 0xfe5f
2524 #define TLSEXT_SIGALG_rsa3072_sphincssha256128frobust 0
      xfe60
2525 #define TLSEXT_SIGALG_sphincsshake256128frobust 0xfe7a
2526 #define TLSEXT_SIGALG_p256_sphincsshake256128frobust 0
      xfe7b
2527 #define TLSEXT_SIGALG_rsa3072_sphincsshake256128frobust 0
      xfe7c
2528 ///// OQS_TEMPLATE_FRAGMENT_DEFINE_SIG_CODE_POINTS_END
```

Algoritmo 12: Definición de algoritmos de firma post-cuánticos.

VIII EXPERIMENTACIÓN: OQS

Este capítulo detalla los siguientes experimentos realizados con OQS-OpenSSL v1.0.2 y v1.1.1:

1. Compilación, análisis y pruebas de OQS-OpenSSL v1.0.2.
2. Análisis de PQCrypto-VPN v1.1 (integración OQS-OpenSSL v1.0.2 con OpenVPN).
3. Compilación, análisis y pruebas de OQS-OpenSSL v1.1.1.
4. Integración y pruebas de OQS-OpenSSL v1.1.1 con Apache.
5. Análisis de la integración OQS-OpenSSL v1.1.1 con Nginx.
6. Compilación, análisis y pruebas de PQCrypto-VPN v1.3 (integración de OQS-OpenSSL v1.1.1 con OpenVPN).

VIII.1 CONSIDERACIONES TÉCNICAS

Las instalaciones, compilaciones y adaptaciones se realizaron en un sistema operativo Lubuntu 19.04 (64 bits) sobre una máquina virtual Qemu/VirtualBox. Se utilizó una máquina virtual para facilitar la portabilidad y duplicación de la infraestructura completa de ser necesario. Debido a que los desarrolladores de las implementaciones analizadas brindan soporte para Ubuntu, se decidió utilizar Lubuntu por ser una variante de la anterior con un menor uso de recursos, lo que facilitó su ejecución en un entorno virtual.

Se creó una estructura de directorios particular para poder realizar las pruebas. Los directorios creados para las pruebas contienen los códigos fuente de las implementaciones, los códigos objeto, binarios ejecutables, las claves y librerías necesarias. A continuación se listan dichos directorios:

- `/opt/openssl102_oqsmaster` para la implementación de liboqs en la v1.0.2 de OQS-OpenSSL
- `/opt/openssl111_oqs` para la implementación de liboqs en la v1.1.1 de OQS-OpenSSL
- `/opt/oqs-demos` para la implementación de las demos de OQS:
  - Integración de OQS-OpenSSL v1.1.1 con Apache
  - Integración de OQS-OpenSSL v1.1.1 con Nginx
- `/opt/openvpn_ms` para la implementación de Openvpn integrada con OQS-OpenSSL v1.1.1**(*)**.
- `/opt/keys` para la centralización de claves privadas, públicas y certificados digitales x509 utilizados durante las pruebas **(*)**. Esta ruta se almacenó en la variable `KEYS` en los módulos de entorno.

Las implementaciones cliente y servidor de cada protocolo se corrieron en el mismo sistema, con excepción de las pruebas de conexión de PQCrypto-VPN, para los cuales se replicó la estructura de directorios de la implementación compilada, las librerías compiladas y el directorio de claves, a un segundo sistema GNU/Linux. Particularmente se utilizó Arch Linux, aunque las características de este sistema no influyen en el resultado de las pruebas ya que se trata de otro sistema GNU/Linux de 64 bits, y no se utilizaron bibliotecas compartidas que dependan del sistema operativo. Los elementos marcados con (*) en el listado anterior representan los directorios y archivos copiados a este segundo sistema GNU/Linux.

Debido a que los experimentos requieren del uso de diferentes versiones de una misma aplicación, como ser OQS-OpenSSL u OpenVPN, se realizaron las compilaciones en los directorios mencionados, pero no se instalaron en el sistema operativo, evitando así conflictos entre las diferentes versiones. Con la intención de facilitar la ejecución de cada binario en su directorio de compilación se configuraron módulos de entorno de la shell (env-modules).

Los paquetes necesarios para realizar las pruebas pueden verse en la Tabla 2 (nombres de referencia según el repositorio actual de Lubuntu 19.04).

Todos los archivos de configuración creados para cada prueba, incluidas las especificaciones de módulos de entorno, claves generadas, listados de algoritmos soportados y demás archivos auxiliares usados para llevar a cabo los experimentos, se encuentran en el repositorio Git que acompaña al presente trabajo (Córdoba, 2022).

Tabla 2: Paquetes necesarios para realizar las pruebas.

Paquete	Descripción
libaprutil1	Librería portable en tiempo de ejecución de Apache.
libaprutil1-dev	Archivos de cabecera de desarrollo para libaprutil1.
flex	Analizador léxico de reconocimiento de patrones de texto.
bison	Generador de analizadores sintácticos de propósito general perteneciente al proyecto GNU.
cmake	Utilizado para controlar la compilación de las aplicaciones.
gcc	Colección de compiladores del proyecto GNU.
libtool	Script de soporte genérico de librerías que facilita la generación de librerías compartidas.
libssl-dev	Bibliotecas de desarrollo, archivos de cabecera y páginas de manual de libssl y libcrypto, partes de OpenSSL.
make	Utilidad que controla la generación de ejecutables y otros archivos desde los códigos fuentes de una aplicación.
ninja-build	Sistema de construcción de aplicaciones utilizado para compilar algunas implementaciones.
git	Cliente del sistema de control de versiones utilizado para descargar los repositorios de las implementaciones.
docker-ce y docker-ce-cli	Necesarios para correr algunas pruebas basadas en imágenes Docker.

VIII.2 OQS-OPENSSL V1.0.2

Esta sección detallan las etapas necesarias para compilar y realizar una prueba de conexión básica usando OQS-OpenSSL v1.0.2. Para lograr el objetivo se siguió la secuencia mostrada a continuación:

1. Creación del directorio de trabajo.
2. Descarga de la rama master del proyecto OQS-OpenSSL v1.0.2.
3. Descarga de liboqs para su integración con con el OQS-OpenSSL.
4. Compilación de liboqs e instalación dentro de OQS-OpenSSL.
5. Compilación de OQS-OpenSSL.
6. Prueba de concepto y análisis de tráfico de red.

Como se mencionó en la Sección VII, la versión 1.0.2 de OpenSSL es obsoleta y no registrará nuevas actualizaciones de mantenimiento, por lo que se utilizó una rama de liboqs compatible con esta versión para lograr su integración correctamente durante las pruebas de concepto.

El Alg. 13 muestra la secuencia de comandos empleada para clonar, desde los repositorios Git, OQS-OpenSSL y liboqs, y luego compilar e instalar ambas implementaciones. Aquí se aprecia el subcomando `checkout` de `git` para moverse, dentro del repositorio de liboqs, a la versión compatible para las pruebas. Se le pasó por argumento el número de identificación de dicha versión. Pueden verse también los comandos necesarios para la configuración, compilación e instalación de esta implementación. El modificador `--prefix` pasado como argumento al script `./configure` de liboqs permite especificar el directorio destino para almacenar los archivos resultantes de la compilación, en este caso el directorio `oqs/` dentro de los fuentes de OQS-OpenSSL. El comando `make` compila liboqs, y luego `make install` instala dicha librería dentro de los fuentes de OQS-OpenSSL v1.0.2. Finalmente se ven los comandos para configurar y compilar OQS-OpenSSL.

```
# Directorio de trabajo:
mkdir /opt/openssl102_oqsmaster/
cd /opt/openssl102_oqsmaster/

# Clonado de OQS-OpenSSL v1.0.2
git clone --branch OQS-OpenSSL_1_0_2-stable \
  https://github.com/open-quantum-safe/openssl.git

# Clonado de liboqs y cambio de rama en git:
git clone --branch master \
https://github.com/open-quantum-safe/liboqs.git
cd liboqs && git checkout \
  ac03b344679ffec6666376c1d955e1c7e30937e3

# Configuración, compilacion e instalación de liboqs:
autoreconf -i
./configure --prefix=../openssl/oqs --enable-shared=no \
--with-sha3=c
make -j
make install

# Compilación de OQS-OpenSSL v1.0.2
cd /opt/openssl102_oqsmaster/openssl/
./Configure no-shared linux-x86_64 -lm
make
```

Algoritmo 13: OQS-OpenSSLv1.0.2. Instalación de liboqs.

Como se mencionó en la Sección VIII.1 no se realizó la instalación completa de la suite, en su lugar que se creó un módulo de entorno, cuya configuración puede verse en el Alg. 14. Nótese la definición de las variables de entorno `KEYSDIR` y `APPSDIR`, ya que se usarán en los comandos siguientes.

```
#%Module1.0#####################################################
##
## modules modulefile
##
# File: /usr/share/modules/modulefiles/openssl/1.0.2

set prefix "OQS-OpenSSL"
set version "1.0.2"

set TOPDIR      /opt
set BASEDIR  /opt/openssl102_oqsmaster/openssl
setenv KEYS     $TOPDIR/keys
setenv APPSDIR $BASEDIR/apps
prepend-path  PATH $BASEDIR/apps
```

Algoritmo 14: OQS-OpenSSL v1.0.2. Módulo de entorno.

El Alg. 15 muestra la carga del módulo de entorno, la verificación de versión de `openssl`, y el uso del subcomando `ciphers` de openssl para extraer las cipher suites habilitadas durante la compilación. El comando `grep OQS` filtra, de dicha salida, únicamente cipher suites las post-cuánticas, aquellas que contienen la cadena `OQS` en su nombre. Por claridad se recortó la salida resaltando la información de interés, la salida completa puede encontrarse en el repositorio Git de la tesis (Córdoba, 2022).

```
module load openssl/1.0.2
openssl version -v
  OpenSSL 1.0.2r  26 Feb 2019

openssl ciphers -V 'ALL' | grep OQS
>
0xFF,0x07- OQSKEM-DEFAULT-ECDHE-ECDSA-AES256-GCM-SHA384 ..
0xFF,0x06- OQSKEM-DEFAULT-ECDHE-RSA-AES256-GCM-SHA384 ..
0xFF,0x03- OQSKEM-DEFAULT-ECDSA-AES256-GCM-SHA384 ..
0xFF,0x02- OQSKEM-DEFAULT-RSA-AES256-GCM-SHA384 ..
0xFF,0x05- OQSKEM-DEFAULT-ECDHE-ECDSA-AES128-GCM-SHA256 ..
0xFF,0x04- OQSKEM-DEFAULT-ECDHE-RSA-AES128-GCM-SHA256 ..
0xFF,0x01- OQSKEM-DEFAULT-ECDSA-AES128-GCM-SHA256 ..
0xFF,0x00- OQSKEM-DEFAULT-RSA-AES128-GCM-SHA256 ..
```

Algoritmo 15: OQS-OpenSSL v1.0.2. Versión y cipher suites OQS.

Las cipher suites listadas incluyen tanto aquellas con algoritmos puramente post-cuánticos (0xff00 - 0xff03), como sus variantes híbridas (0xff04 - 0xff07). Estos últimos combinan los algoritmos post-cuánticos con un intercambio de claves basado en ECDH. Las definiciones de estas cipher suites se encuentra en el archivo `openssl/ssl/tls1.h` del código fuente, tal y como puede verse en la Fig. 7.

```
566 /* OQS_KEM based ciphersuites */
567 # define TLS1_CK_OQSKEM_DEFAULT_RSA_WITH_AES_128_GCM_SHA256               0x0300FF00
568 # define TLS1_CK_OQSKEM_DEFAULT_ECDSA_WITH_AES_128_GCM_SHA256            0x0300FF01
569 # define TLS1_CK_OQSKEM_DEFAULT_RSA_WITH_AES_256_GCM_SHA384               0x0300FF02
570 # define TLS1_CK_OQSKEM_DEFAULT_ECDSA_WITH_AES_256_GCM_SHA384            0x0300FF03
571 # define TLS1_CK_OQSKEM_DEFAULT_ECDHE_RSA_WITH_AES_128_GCM_SHA256        0x0300FF04
572 # define TLS1_CK_OQSKEM_DEFAULT_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256      0x0300FF05
573 # define TLS1_CK_OQSKEM_DEFAULT_ECDHE_RSA_WITH_AES_256_GCM_SHA384        0x0300FF06
574 # define TLS1_CK_OQSKEM_DEFAULT_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384      0x0300FF07
```

Figura 7: OQS-OpenSSL 1.0.2. Definición de Cipher suites.

Las pruebas iniciales se llevaron a cabo utilizando clave privada RSA de 2048b, y certificado digital x509 autofirmado. El Alg. 16 muestra los comandos utilizados para generar estos elementos, ejecutar el servidor TLS (subcomando `s_server` de OpenSSL), y ejecutar el cliente (subcomando `s_client` de OpenSSL) para establecer la conexión.

```
# Generación de clave y certificado autofirmado:
openssl req -x509 -new -newkey rsa:2048 \
-keyout $KEYSDIR/server_rsa.key \
-out $KEYSDIR/server_rsa.crt -nodes \
-subj "/CN=oqstest rsa2048" -days 365 \
-config $APPSDIR/openssl.cnf

# Ejecución del servidor TLS (defecto puerto TCP 4433):
openssl s_server -cert $KEYSDIR/server_rsa.crt \
-key $KEYSDIR/server_rsa.key -www

# Conexión del cliente:
module load openssl/1.0.2  # env-module en otra shell
openssl s_client -CAfile $KEYSDIR/server_rsa.key \
-cipher OQSKEM-DEFAULT -connect localhost:4433
```

Algoritmo 16: OQS-OpenSSLv1.0.2. Generación de clave y certificado.

La ejecución del cliente muestra que se forzó el uso de una cipher suite `OQSKEM-DEFAULT` utilizando el modificador `--cipher`. La conexión se estableció satisfactoriamente, y ambos nodos negociaron la cipher suite correcta para TLSv1.2, como puede verse en la Fig. 8.

VIII.2.1 Compilación con algoritmos OQS no híbridos

Las pruebas anteriores se realizaron con la implementación OQS-OpenSSL compilada utilizando algoritmos post-cuánticos híbridos, opción recomendada por los desarrolladores del proyecto. En este apartado se detalla la construcción de OQS-OpenSSL v1.0.2 con cipher suites puramente post-cuánticas con la intención de verificar su usabilidad. Para ello se habilitó la macro `OPENSSL_NO_HYBRID_OQSKEM_ECDHE` durante la configuración de la suite previo a su compilación. El Alg. 17 muestra esta configuración al momento de compilar, y lista las cipher suites resultantes, todas en TLS v1.2.

```
root@juncotic-lubuntu:~# openssl s_server -cert $KEYSDIR/server_rsa.crt -key $KEYSDIR/server_rsa.key -www
Using default temp DH parameters
ACCEPT
ACCEPT
```

```
---
New, TLSv1/SSLv3, Cipher is OQSKEM-DEFAULT-ECDHE-RSA-AES256-GCM-SHA384
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : OQSKEM-DEFAULT-ECDHE-RSA-AES256-GCM-SHA384
    Session-ID: 25D3F0FA1ADDD8B79C89E09607A30F03BA63EFA3773A1519CE801C8F7C5F0B0F
    Session-ID-ctx:
    Master-Key: 1E08AF7A87FF9F1E7F46791D4CE88CBF802C3AF3B70B3B67747A3A579D2E17B413F3F6A35A1521EE51074FD5F8
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 300 (seconds)
    TLS session ticket:
    0000 - 76 1e 9c 1f 15 7c 1f 33-eb 4a e4 61 bb c8 b9 8f   v....|.3.J.a....
    0010 - 3a af 16 f1 81 3d e5 29-61 bf e7 b2 25 67 87 0a   :....=.)a...%g..
```

Figura 8: Cipher suite OQSKEM negociada.

```
# Compilación sin ciphe rsuites híbridas:
cd /opt/openssl102_oqsmaster/openssl/
./Configure -DOPENSSL_NO_HYBRID_OQSKEM_ECDHE no-shared
   linux-x86_64 -lm
make

# Cihpersuites post-cuánticas puras:
module load openssl/1.0.2
openssl ciphers -V 'ALL'|grep OQS
>
0xFF,0x03 - OQSKEM-DEFAULT-ECDSA-AES256-GCM-SHA384 ...
0xFF,0x02 - OQSKEM-DEFAULT-RSA-AES256-GCM-SHA384 ...
0xFF,0x01 - OQSKEM-DEFAULT-ECDSA-AES128-GCM-SHA256 ...
0xFF,0x00 - OQSKEM-DEFAULT-RSA-AES128-GCM-SHA256 ...
```

Algoritmo 17: OQS-OpenSSLv1.0.2. Cipher suites post-cuánticas puras.

Como se puede apreciar, únicamente se incluyeron los algoritmos 0xFF00 hasta el 0xFF03, no las variantes híbridas que se veían en el Alg. 15. Durante la prueba de conexión el cliente ofreció sólo las cipher suites post-cuánticas, omitiendo las híbridas y las tradicionales. Para verificar este caso se tomó una captura de tráfico de red con Wireshark (Fig. 9). Aquí pueden verse las cipher suites soportadas por el cliente y ofrecidas al servidor mediante el mensaje `Client Hello` de la negociación TLS v1.2.

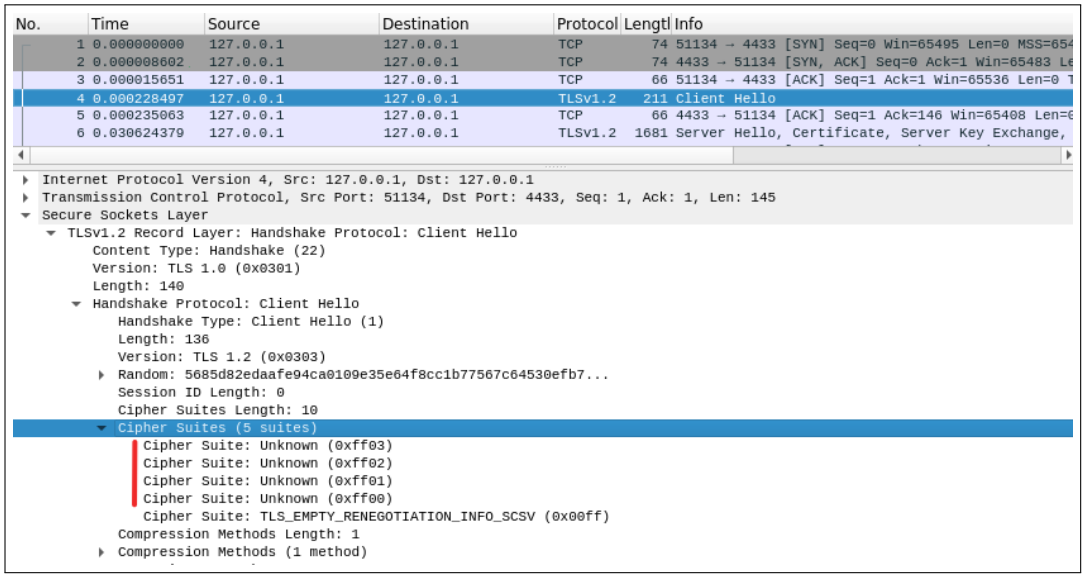

Figura 9: Captura de cipher suite en OQS no híbrido.

VIII.3 OQS-OPENSSL v1.0.2 Y OPENVPN

En esta sección se detallan las pruebas realizadas con la implementación PQCrypto-VPN (Microsoft) en su versión v1.1, que integra OQS-OpenSSL v1.0.2 con OpenVPN. Si bien la versión v1.0.2 de OpenSSL se considera obsoleta y PQCrypto-VPN v1.1 está discontinuada al momento de escribir este trabajo, resulta de interés analizar un ejemplo de uso de cipher suite post-cuántica durante la negociación de comunicación OpenVPN, ya que dichas negociación no está presente en la v1.1.1 de OQS-OpenSSL.

Las pruebas que se mencionan en este apartado se realizaron durante el año 2019 cuando OpenVPN v1.0.2 tenía soporte y mantenía compatibilidad con PQCrypto-VPN, por lo que el análisis se centra en los experimentos realizados y los resultados obtenidos, omitiendo los pasos de descarga y compilación de las implementaciones de software. La Sección VIII.7 analiza PQCrypto-VPN v1.3, que combina OpenVPN con OQS-OpenSSL v1.1.1.

Con la intención de facilitar la ejecución de binarios y carga de librerías, se creó un archivo de configuración de módulo de entorno con el contenido

mostrado en el Alg. 18. Nótese la variable `LD_LIBRARY_PATH` utilizada para cargar las librerías propias de la compilación. La variable `CONFDIR` facilitó la ubicación de las configuraciones.

```
#%Module1.0#####################################################
##
## modules modulefile
##
## File: /usr/share/modules/modulefiles/pqcryptovpn/1.1

set prefix "PQCrypto-VPN"
set version "1.1"

set TOPDIR      /opt
set BASEDIR  /opt/openvpn_ms/PQCryptoVPN_1.1/openvpn/build
   /stage/usr/local/openvpn
setenv KEYSDIR  $TOPDIR/keys
setenv LD_LIBRARY_PATH $BASEDIR/lib
setenv BASEDIR $BASEDIR
setenv CONFDIR $BASEDIR/etc/config

prepend-path  PATH $BASEDIR/sbin
```

Algoritmo 18: PQCryptoVPN v1.1. Módulo de entorno.

El Alg. 19 muestra la carga del módulo de entorno y la verificación de versión de `openvpn`. Este Algoritmo también muesrta el uso del modificador `--show-tls` utilizado para listar las cipher suites TLS disponibles.

```
# Módulo de entorno:
module load pqcryptovpn/1.1

# Verificación de versión:
openvpn --version
>
liboqs 1.0.0 [git:pqcrypto/65fd62a9e8038cf2] x86_64-pc-
   linux-gnu [SSL (OpenSSL)] [LZO] [LZ4] [EPOLL] [MH/
   PKTINFO] [AEAD] built on Apr 14 2020
library versions: OpenSSL 1.0.2m-dev  xx XXX xxxx, LZO
   2.10
Originally developed by James Yonan
[...]

#Lista de cipher suites disponibles:
openvpn --show-tls
```

Algoritmo 19: PQCryptoVPN v1.1. Versión y Cipher suites.

Las cipher suites incluidas corresponden a TLS v1.2 puesto que la seguridad de PQCryptoVPN v1.1 se basa en OQS-OpenSSL v1.0.2. La siguiente

lista resume las cipher suites post-cuánticas e híbridas. La salida del comando se omitió por motivos de claridad y extensión. La salida completa puede encontrarse en el repositorio Git de la tesis (Córdoba, 2022).

- OQSKEX-**<S>**-PICNIC-AES256-GCM-SHA384, con **<S>**: SIDH-MSR, RLWE-MSRLN16 y LWE-FRODO-RECOMMENDED
- OQSKEX-**<S>**-ECDSA-WITH-AES-256-GCM-SHA384 y OQSKEX-**<S>**-RSA-WITH-AES-256-GCM-SHA384, con **<S>**: MLWE-KYBER, NTRU, CODE-MCBITS y SIDH-IQC-REF
- OQSKEX-**<S>**-ECDSA-AES256-GCM-SHA384 y OQSKEX-**<S>**-RSA-AES256-GCM-SHA384, con **<S>**: SIDH-MSR, RLWE-MSRLN16, LWE-FRODO-RECOMMENDED, RLWE-NEWHOPE, RLWE-BCNS15 y GENERIC
- OQSKEX-**<S>**-ECDSA-WITH-AES-128-GCM-SHA256 y OQSKEX-**<S>**-RSA-WITH-AES-128-GCM-SHA256, con **<S>**: MLWE-KYBER, NTRU, CODE y SIDH
- OQSKEX-**<S>**-ECDSA-AES128-GCM-SHA256 y OQSKEX-**<S>**-RSA-AES128-GCM-SHA256, con **<S>**: SIDH-MSR, RLWE-MSRLN16, LWE-FRODO-RECOMMENDED, RLWE-NEWHOPE, RLWE-BCNS15 y GENERIC

La opción `tls-cipher` de las configuraciones de OpenVPN permite habilitar las cipher suites específicas de la lista anterior. Se realizó la prueba de concepto usando una cipher suite que incluya el protocolo NTRU por su buen rendimiento (Baktu, 2017). La cipher suite seleccionada es la siguiente:

```
OQSKEX-NTRU-ECDHE-RSA-WITH-AES-256-GCM-SHA384
```

Debido a que esta versión de OpenVPN no utiliza intercambio de claves post-cuánticos, se procedió a desactivar el parámetro `ecdh-curve`. Las configuraciones resultantes, tanto para el servidro como para el cliente, pueden verse en el Alg. 20.

```
# Archivo $CONFDIR/server.ovpn
ca /opt/keys/sample-keys/ca.crt
cert $KEYSDIR/sample-keys/server.crt
key $KEYSDIR/sample-keys/server.key
dh $KEYSDIR/sample-keys/dh2048.pem
tls-cipher OQSKEX-NTRU-ECDHE-RSA-WITH-AES-256-GCM-SHA384
; ecdh-curve

# Archivo $CONFDIR/client.ovpn
remote 192.168.0.20 1194     # dirección IP del servidor
ca $KEYSDIR/sample-keys/ca.crt
cert $KEYSDIR/sample-keys/client.crt
key $KEYSDIR/sample-keys/client.key
tls-cipher OQSKEX-NTRU-ECDHE-RSA-WITH-AES-256-GCM-SHA384
; ecdh-curve
```

Algoritmo 20: PQCryptoVPN v1.1. Archivos de configuración.

Para verificar la conexión se replicó el directorio `PQCryptoVPN/` de servidor a cliente, por lo que las rutas relativas utilizadas coinciden en ambos casos. Esto permitió el uso del módulo de entorno. Para la prueba de concepto se lanzó el binario de `openvpn` invocando la configuración del servidor en uno de los equipos, y del cliente en el otro, como puede verse en el Alg. 21.

```
# Ejecución del servidor:
module load pqcryptovpn/1.1
openvpn --config $CONFDIR/server.ovpn

# Ejecución del cliente (otra shell):
module load pqcryptovpn/1.1
openvpn --config $CONFDIR/client.ovpn
```

Algoritmo 21: PQCryptoVPN v1.1. Ejecución.

La cipher suite NTRU elegida pudo ser negociada satisfactoriamente. Las Figs. 10 y 11 muestran la terminal del servidor y del cliente respectivamente.

```
Thu Sep 24 21:21:16 2020 192.168.0.10:54669 peer info: IV_COMP_STUB=1
Thu Sep 24 21:21:16 2020 192.168.0.10:54669 peer info: IV_COMP_STUBv2=1
Thu Sep 24 21:21:16 2020 192.168.0.10:54669 peer info: IV_TCPNL=1
Thu Sep 24 21:21:16 2020 192.168.0.10:54669 Control Channel: TLSv1.2, cipher TLSv1/SSLv3
OQSKEX-NTRU-ECDHE-RSA-WITH-AES-256-GCM-SHA384, 2048 bit RSA
Thu Sep 24 21:21:16 2020 192.168.0.10:54669 [Test-Client] Peer Connection Initiated with
[AF_INET]192.168.0.10:54669
Thu Sep 24 21:21:16 2020 Test-Client/192.168.0.10:54669 MULTI_sva: pool returned IPv4=10.
8.0.6, IPv6=(Not enabled)
```

Figura 10: PQCrypto-VPN. Cipher suite post-cuántica (servidor).

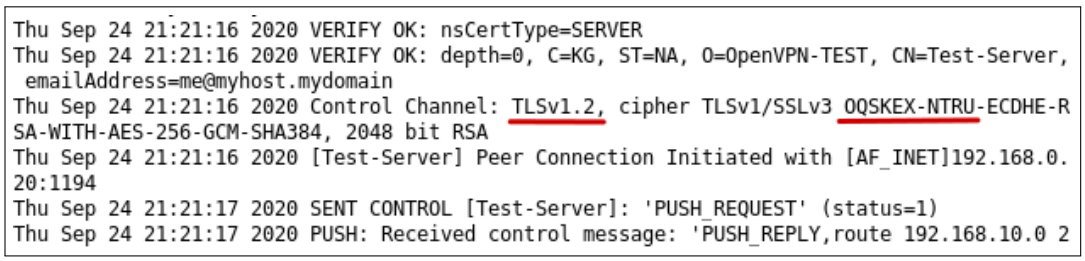

Figura 11: PQCrypto-VPN. Cipher suite post-cuántica (cliente).

VIII.4 OQS-OPENSSL v1.1.1

A continuación se detallan los experimentos llevados a cabo con la versión 1.1.1 de OQS-OpenSSL. Estos consistieron en:

1. Descarga de los códigos fuente.
2. Compilación de liboqs.
3. Compilación de OpenSSL integrado con liboqs.
4. Configuración de módulo de entorno.
5. Verificación de algoritmos reconocidos por la implementación.
6. Análisis de TLS con algoritmos asimétricos tradicionales.
7. Análisis de algoritmos KEM en TLS.
8. Generación y análisis de claves y certificados post-cuánticos.
9. Análisis de TLS con algoritmos asimétricos post-cuánticos.
10. Incorporación de algoritmos KEM post-cuánticos en TLS.

Para montar la prueba de concepto se creó un directorio de trabajo y se clonó la rama estable desde los repositorios Git del proyecto. Dentro del directorio de los códigos fuente de OQS-OpenSSL se clonó la biblioteca liboqs, se compiló e instaló. La secuencia de comandos puede verse en el Alg. 22. Nótese que se especificó como destino de instalación de liboqs el directorio `oqs/` dentro de los fuentes de OQS-Openssl previamente clonado. Este algoritmo muestra también la compilación de OQS-OpenSSL.

```
# Clonado de OQS-OpenSSL v1.1.1:
mkdir /opt/openssl111_oqs/
cd /opt/openssl111_oqs/
git clone --branch OQS-OpenSSL_1_1_1-stable \
https://github.com/open-quantum-safe/openssl.git

# Clonado de liboqs:
cd /opt/openssl111_oqs/openssl/
git clone --branch \
master https://github.com/open-quantum-safe/liboqs.git

# Compilación de liboqs:
cd liboqs
mkdir build && cd build
cmake -GNinja \
-DCMAKE_INSTALL_PREFIX=/opt/openssl111_oqs/oqs ..
ninja
ninja install

# Compilación de OpenSSL:
cd /opt/openssl111_oqs/openssl/
./Configure no-shared linux-x86_64 -lm
make
```

Algoritmo 22: OQS-OpenSSLv1.1.1. Compilación.

Debe resaltarse el comando `cmake` utilizado para compilar liboqs. Como se comentó en la Sección VII.2.1, aquí pueden definirse macros para activar o desactivar algoritmos KEM o de firma digital. Si no se especifica ninguna macro, se habilitarán todos los algoritmos detallados en las secciones VII.2.1 y VII.2.2. Para poder desactivar un algoritmo particular puede utilizarse la opción `-DOQS_ENABLE_<T>_<ALG>`, donde `<T>` puede ser `KEM` o `SIG`, y `<ALG>` representa el algoritmo en cuestión. Estas macros soportan dos valores: ON y OFF. Por ejemplo, para desactivar el algoritmo KEM SIKE puede añadirse la siguiente opción a la llamada a `cmake`:

```
-DOQS_ENABLE_KEM_SIKE=OFF
```

Al momento de compilar liboqs `cmake` cargará los algoritmos habilitados en la cabecera `openssl/oqs/include/oqs/oqsconfig.h`, de modo que aquellos que sean habilitados aquí estarán disponibles en OpenSSL luego.

La compilación generó los binarios ejecutables necesarios para realizar las pruebas. No se instaló OQS-OpenSSL en el sistema (`make install`) para evitar conflictos con otras instalaciones de OpenSSL. En su lugar se creó un módulo de entorno para facilitar su uso, como se muestra en el Alg. 23.

```
#%Module1.0#####################################################
##
## modules modulefile
##
## File: /usr/share/modules/modulefiles/openssl/1.1.1

set prefix "OQS-OpenSSL"
set version "1.1.1"

set TOPDIR      /opt
set BASEDIR  /opt/openssl111_oqs/openssl
setenv KEYSDIR  $TOPDIR/keys
setenv APPSDIR $BASEDIR/apps

prepend-path  PATH $BASEDIR/apps
```

Algoritmo 23: OQS-OpenSSL v1.1.1. Módulo de entorno.

Como se comentó en la Sección VII.2 esta versión no posee cipher suites post-cuánticas, por lo que no se realizaron mayores análisis en este punto. No obstante, a diferencia de la versión anterior, aquí sí existe soporte para algoritmos post-cuánticos de firma digital y de cifrado asimétrico. El Alg. 24 muestra la carga del módulo de entorno, la verificación de versión, y el uso del comando `list` de `openssl` para listar los algoritmos de clave pública. La salida de este último comando se omitió por motivos de claridad y extensión. La salida completa puede encontrarse en el repositorio Git de la tesis (Córdoba, 2022). Pudo constatarse que los algoritmos listados corresponden con los documentados en la Sección VII.2.2.

```
# Carga del móudlo de entorno:
module load openssl/1.1.1
# Versión utilizada:
openssl version
> OpenSSL 1.1.1m  14 Dec 2021, Open Quantum Safe 2022-01
# Algoritmos de clave pública:
openssl list -public-key-algorithms
```

Algoritmo 24: OQS-OpenSSL v1.1.1. Versión y cifradores.

Las pruebas se llevaron a cabo utilizando las claves RSA creadas previamente en la Sección VIII.2: `server_rsa.key` y `server_rsa.crt`. El Alg. 25 muestra la ejecución del servidor y del cliente.

```
# Ejecución del servidor: (TCP 4433)
module load openssl/1.1.1
openssl s_server -cert $KEYSDIR/server_rsa.crt \
-key $KEYSDIR/server_rsa.key -www

# Ejecución del cliente:
module load openssl/1.1.1 # en otra shell
openssl s_client -connect localhost:4433
```

Algoritmo 25: OQS-OpenSSLv1.1.1. Ejecución del servidor.

A diferencia del la prueba de OpenSSL v1.0.2, durante el experimento se negoció una cipher suite tradicional de OpenSSL en TLS v1.3 de forma predeterminada, como puede verse en la captura de la Fig. 12. En esta captura puede apreciarse además el uso de X25519 como algoritmo utilizado de intercambio de claves (ECDH con curva Curve25519).

```
root@juncotic-lubuntu:~# openssl s_server -cert $KEYSDIR/server_rsa.crt \
> -key $KEYSDIR/server_rsa.key -www
Using default temp DH parameters
ACCEPT
issuer=C = AR, ST = server rsa, O = Internet Widgits Pty Ltd

---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: RSA-PSS
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 1435 bytes and written 483 bytes
Verification error: self signed certificate
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
```

Figura 12: Intercambio OpenSSL v1.1.1.

Como se comentó en los párrafos previos, esta versión implementa diversos algoritmos de firma digital post-cuánticos. Si se analiza la captura del mensaje `Client Hello` de la negociación TLS, dentro de las extensiones de TLS1.3 pueden verse los algoritmos de firma ofrecidos por el cliente, tal y como puede apreciarse en la captura de tráfico de la Fig. 13.

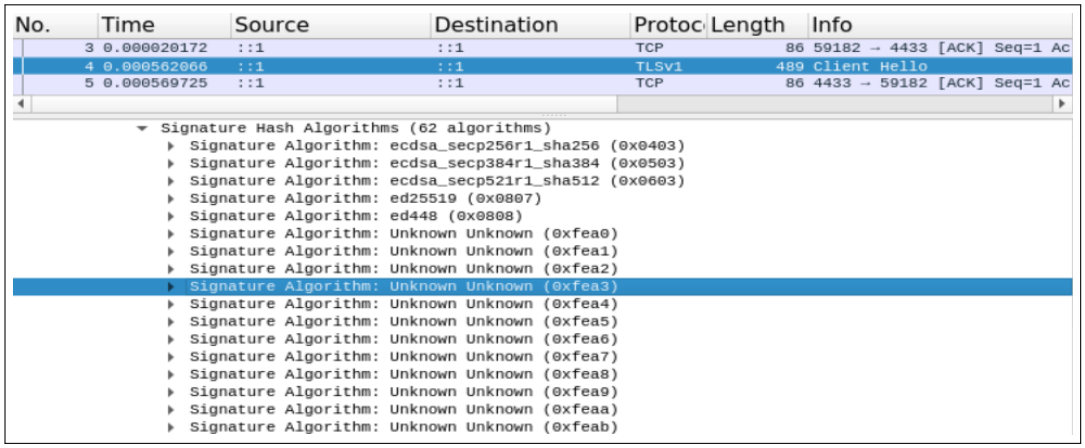

Figura 13: Algoritmos de firma en OpenSSL v1.1.1.

La mayoría de los sniffers de tráfico de red, solamente reconocen algoritmos estándar, es decir, capturan los códigos de identificación de los algoritmos que viajan por la red, y *mapean* dichos códigos con nombres de algoritmos estándar. Particularmente Wireshark, cuando no reconoce un algoritmo, lo etiqueta como `Unknown` en su interfaz de usuario. Junto a esta etiqueta agrega el código de identificación del algoritmo que ha capturado. Los códigos mostrados en la Fig. 13 representan los algoritmos ofrecidos por el cliente durante el intercambio TLS del presente experimento. El archivo `./openssl/ssl/ssl_local.h` define, tal y como se comentó en la Sección VII.2.2, los nombres de estos algoritmos, y los asocia directamente con los códigos capturados por Wireshark. Por ejemplo, el algoritmo marcado arbitrariamente en la Fig. 13, `0xfea3` corresponde con Dilithium3.

TLS con claves y certificados post-cuánticos.

El siguiente experimento consistió en analizar la conexión TLS utilizando certificado y clave asimétricos basados en un algoritmo post-cuántico soportado. Para llevar a cabo los experimentos se seleccionó el algoritmo asimétrico Rainbow (Ding y Schmidt, 2005), uno de los finalistas de la tercer ronda del proceso de estandarización del NIST (NIST, 2018). Este algoritmo

provee implementaciones en liboqs para los niveles L1 y L5 de estandarización. Las variantes de Rainbow soportadas se caracterizan por el gran tamaño de las claves en comparación con otros finalistas, lo que resulta de especial interés para realizar los experimentos ya que permite analizar conexiones TLS con un flujo relativamente grande de tráfico. Particularmente Rainbow-V-Classic, el algoritmo post-cuántico puro de nivel de seguridad L5, posee una clave privada de 15444800 bits.

El algoritmo Rainbow y sus variantes soportadas por OQS-OpenSSL fueron mencionados en la Sección VII.2.2. Como se comentó oportunamente, estos algoritmos incluyen variantes post-cuánticas puras y variantes híbridas. Rainbow particularmente es un algoritmo que posee claves extremadamente grandes comparadas con las de otros algoritmos del mismo nivel de seguridad del NIST. Dentro de OQS-OpenSSL y liboqs Rainbow provee dos implementaciones post-cuánticas puras, Rainbow-I-Classic y Rainbow-V-Classic, con niveles de seguridad L1 y L5 respectivamente. Liboqs incorpora tres variantes híbridas con ECDSA, dos para la versión de nivel L1 (curvas p256 y rsa3072) y una para la versión de nivel L5 (curva p521).

La creación de la clave privada y certificado digital autofirmado Rainbow-V-Classic puro se realizó como muestra el Alg. 26.

```
module load openssl/1.1.1
openssl req -x509 -nodes -days 1000 \
  -newkey rainbowVclassic \
  -keyout $KEYSDIR/rainbowVclassic.key \
  -out $KEYSDIR/rainbowVclassic.crt \
  -subj "/CN=oqstest RainbowVclassic"
```

Algoritmo 26: OQS-OpenSSLv1.1.1. Generación de clave y certificado.

Aquí cabe resaltar el “asunto” o *subject* especificado en el certificado x509 mediante la opción `-subj: /CN=oqstest RainbowVclassic`, ya que esta información será visible al analizar el contenido del certificado. La validez del mismo se estableció en 1000 días. Los archivos generados fueron:

- Clave Privada: `$KEYSDIR/rainbowVclassic.key`
- Certificado Digital: `$KEYSDIR/rainbowVclassic.crt`

Puede usarse el mismo binario `openssl` para extraer la información del certificado digital y verificar los datos almacenados en el mismo. El comando y la salida resumida pueden verse en el Alg. 27 (la salida completa se encuentra en el repositorio Git que acompaña esta tesis (Córdoba, 2022)).

```
# Extracción de datos:
module load openssl/1.1.1
openssl x509 -in $KEYSDIR/rainbowVclassic.crt -noout -text

# Salida (resumida para resaltar información de interés)
Certificate:
[...]
  Signature Algorithm: rainbowVclassic
  Issuer: CN = oqstest RainbowVclassic
  Validity
    Not Before: Mar  1 14:22:10 2022 GMT
    Not After : Nov 25 14:22:10 2024 GMT
  Subject: CN = oqstest RainbowVclassic
  Subject Public Key Info:
    Public Key Algorithm: rainbowVclassic
    rainbowVclassic Public-Key:
    [...]
  Signature Alg4orithm: rainbowVclassic
```

Algoritmo 27: OQS-OpenSSLv1.1.1. Datos del certificado.

El algoritmo de clave asimétrica utilizado es Rainbow-V-Classic, y al tratarse de un certificado autofirmado, la firma digital se generó utilizando el mismo algoritmo. Este juego de certificado y clave puede ser utilizado para realizar pruebas de concepto de conexión cliente-servidor de la misma manera que antes se realizó con RSA. A modo de verificación se ejecutó servidor y cliente como muestra el Alg. 28.

```
# Ejecución del servidor: (TCP 4433)
module load openssl/1.1.1
openssl s_server -cert $KEYSDIR/rainbowVclassic.crt \
  -key $KEYSDIR/rainbowVclassic.key -www

# Ejecución del cliente (otra shell):
module load openssl/1.1.1
openssl s_client -connect localhost:4433
```

Algoritmo 28: OQS-OpenSSLv1.1.1. Prueba usando Rainbow-V-Classic.

La Fig. 14 muestra, en la terminal superior, la ejecución del servidor, y en la inferior la del cliente.

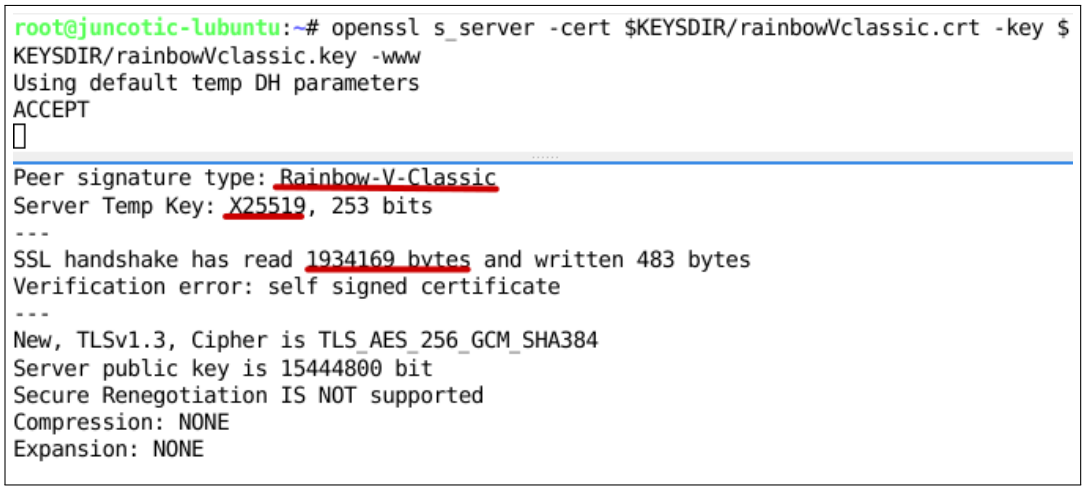

Figura 14: Conexión cliente-servidor usando Rainbos-V-Classic.

La negociación TLSv1.3 con el cliente fue satisfactoria, lo que permite afirmar que la prueba de TLSv1.3 usando algoritmos de encapsulamiento de clave y de firma digital post-cuánticos fue un éxito. Como puede verse en esta figura, el tamaño de la clave recibida por el cliente es de 15444800 bits, cumplimentando un total de 1934169 bytes leídos por el cliente durante el handshake TLS. Además se ve que el algoritmo KEM de intercambio de claves es ECDH utilizando la curva Curve25519, un algoritmo tradicional.

Incorporación de KEM post-cuántico.

El siguiente paso fue incorporar un algoritmo de intercambio de claves post-cuántico además del algoritmo de autenticación y firma digital. Para ello se verificó la conexión con los algoritmos documentados en la Sección VII.2.1 y la misma pudo establecerse sin inconvenientes. Para la prueba de concepto se seleccionó de manera arbitraria Kyber1024, un algoritmo KEM post-cuántico finalista del proceso de estandarización del NIST. El Alg. 29 muestra el comando utilizado por el cliente para establecer la conexión, mientras que la Fig. 15 permite apreciar la terminal del cliente, donde puede verse

el uso de este algoritmo. Esta conexión utiliza un algoritmo post-cuántico tanto para el intercambio de claves (KEM) en TLS, como para autenticación usando criptografía asimétrica.

```
# Ejecución del cliente (otra shell):
module load openssl/1.1.1
openssl s_client -connect localhost:4433 -curves kyber1024
```

Algoritmo 29: OQS-OpenSSLv1.1.1. Uso de KEM post-cuántico.

```
Peer signature type: Rainbow-V-Classic
Server Temp Key: kyber1024
---
SSL handshake has read 1935753 bytes and written 1979 bytes
Verification error: self signed certificate
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 15444800 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
```

Figura 15: Conexión cliente-servidor incorporando KEM Kyber1024.

Verificación de firma digital post-cuántica.

Con la intención de realizar una conexión TLS con seguridad post-cuántica completa, solamente restaba lograr la verificación del certificado digital del servidor mediante la incorporación de una autoridad certificante. Este paso eliminaría el error de verificación que se aprecia en la Fig. 15, error debido al uso de un certificado autofirmado. Para cumplimentar este objetivo, y para analizar la combinación de algoritmos asimétricos post-cuánticos en la comunicación TLS, se llevaron a cabo los siguientes pasos:

1. Generación de certificado y clave privada de la CA.
2. Generación de un CSR (Certificate Sign Request - Solicitud de firma de certificado) a partir de la clave privada Rainbow-V-Classic.
3. Generación de un certificado digital x509 basado en Rainbow-V-Classic firmado por la clave privada de la CA.

4. Ejecución del servidor utilizando dicho certificado firmado.
5. Conexión del cliente para verificar la firma.

Como algoritmo asimétrico post-cuántico para la autoridad certificante se eligió arbitrariamente otro de los finalistas del NIST: Falcon, en su variante Falcon1024. El Alg. 30 muestra los comandos que generan las claves Falcon1024 para firmar el certificado Rainbow-V-Classic, y los comandos de ejecución de servidor y cliente.

```
module load openssl/1.1.1

#Generacion de clave y certificado Falcon1024
openssl req -x509 -nodes -days 1000 -newkey falcon1024 \
-keyout $KEYSDIR/falcon1024.key \
-out $KEYSDIR/falcon1024.crt \
-subj "/CN=oqstest Falcon 1024"

# Creacion del CSR basado en la clave privada
# Rainbow-V-Classic generada previamente
openssl req -new -key $KEYSDIR/rainbowVclassic.key \
-out $KEYSDIR/signed_rainbowVC.csr

# Firma del CSR utilizando la clave privada Falcon1024:
openssl x509 -req -days 1000
-in $KEYSDIR/signed_rainbowVC.csr \
-CAkey $KEYSDIR/falcon1024.key \
-CA $KEYSDIR/falcon1024.crt \
-out $KEYSDIR/signed_rainbowVC.crt -set_serial 01

# Ejecucion del servidor:
openssl s_server -cert $KEYSDIR/signed_rainbowVC.crt \
-key $KEYSDIR/rainbowVclassic.key -www

# Ejecucion del cliente
openssl s_client -connect localhost:4433 \
-curves kyber1024 -CAfile $KEYSDIR/falcon1024.crt
```

Algoritmo 30: OQS-OpenSSLv1.1.1. TLS con certificado firmado.

El Alg. 31 muestra los datos relevantes del certificado digital (la salida completa se encuentra en el repositorio Git que acompaña el presente trabajo (Córdoba, 2022)). Puede apreciarse que se trata de un certificado digital RainbowVClassic firmado utilizando Falcon1024.

```
# Extracción de datos:
module load openssl/1.1.1
openssl x509 -in /opt/keys/signed_rainbowVC.crt -noout -
    text

# Salida (resumida para resaltar información de interés)
Certificate:
[...]
  Signature Algorithm: falcon1024
  Issuer: CN = oqstest Falcon 1024
  Validity
    Not Before: Mar  2 23:08:12 2022 GMT
    Not After : Nov 26 23:08:12 2024 GMT
  Subject: C = AR, ST = Mendoza, L = San Rafael, O =
   JuncoTIC, OU = tesis, CN = Diego Cordoba, emailAddress
   = diego@juncotic.com
  Subject Public Key Info:
    Public Key Algorithm: rainbowVclassic
      rainbowVclassic Public-Key:
    [...]
```

Algoritmo 31: OQS-OpenSSLv1.1.1. Datos del cert. firmado.

La Fig. 16 muestra el resultado de la negociación TLS, donde puede apreciarse, en la terminal inferior (cliente), la verificación correcta del certificado recibido desde el servidor. Este experimento permitió lograr una comunicación segura entre cliente y servidor utilizando únicamente mecanismos post cuánticos: un algoritmo KEM para el intercambio de claves (Kyber1024), un algoritmo asimétrico utilizado para autenticación (Rainbow-V-Classic) y un algoritmo asimétrico utilizado para firma digital (Falcon1024). La verificación del certificado digital se llevó a cabo satisfactoriamente, lo que da cuenta de que ambos nodos reconocieron correctamente los formatos y tamaños de claves, y lograron montar una comunicación resistente a ataques cuánticos.

VIII.5 OQS-OPENSSL v1.1.1 Y HTTPS (APACHE)

El proyecto OQS provee una serie de imágenes Docker (Docker Inc., 2013) con versiones pre-compiladas de algunos servicios post-cuánticos (Stebila et al., 2022). Las imágenes Docker permiten crear entornos virtuales aislados, denominados contenedores. Estos contenedores le ofrecen al usuario entornos repetibles de construcción y prueba de aplicaciones. Particular-

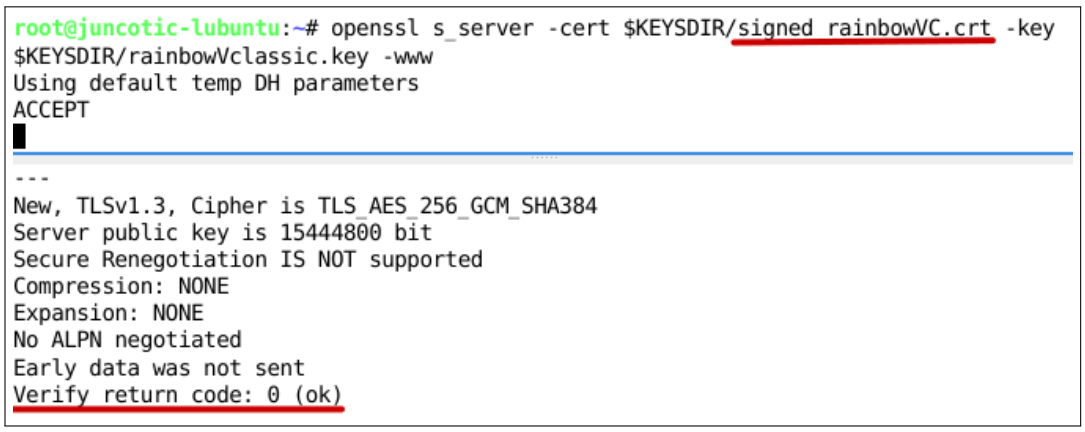

Figura 16: Conexión cliente-servidor con certificado firmado.

mente en esta tesis los contenedores Docker facilitaron la realización de los experimentos con las implementaciones tal y como los desarrolladores las ofrecían, sin necesidad de compilaciones y configuraciones que dependan de detalles propios de cada sistema operativo.

Los experimentos llevados a cabo en esta sección son los siguientes:

1. Construcción el contenedor Docker de Apache.
2. Análisis de sus configuraciones predeterminadas, y adaptaciones.
3. Análisis de conexión con parámetros predeterminados.
4. Extracción y análisis de claves y certificados digitales del contenedor.
5. Análisis del binario de OpenSSL integrado en la imagen Docker.
6. Incorporación de algoritmos KEM puros e híbridos en TLS.
7. Incorporación de un cliente cURL post-cuántico.
8. Análisis de conexión TLS post-cuántica (autenticación y KEM).

Contenedor Docker de Apache.

Una de las demos Docker provistas por OQS corresponde con una implementación de Apache integrado con OQS-OpenSSL v1.1.1. Uno de los parámetros importantes de la imagen Docker es el algoritmo de firma y autenticación utilizado de manera predeterminada a la hora de crear el conte-

nedor. Puede cambiarse el algoritmo de autenticación predeterminado utilizando el modificador `--build-alg SIG_ALG=<SIG>` en Docker, donde `<SIG>` representa un algoritmo válido (véase Sección VII.2.2). El Alg. 32 muestra el comando necesario para clonar el repositorio y crear el contenedor, ya sea utilizando el algoritmo de firma predeterminado como haciendo uso de `--build-alg`. Se muestra como último comando el necesario para ejecutar el contenedor previamente creado.

```
# Clonado del repositorio oficial (recomendado):
mkdir /opt/oqs-demos && cd /opt/oqs-demos/
git clone \
  https://github.com/open-quantum-safe/oqs-demos.git

# Clonado del repositorio (fork modificado):
mkdir /opt/oqs-demos && cd /opt/oqs-demos/
git clone git@github.com:d1cor/oqs-demos.git
# Cambio de branch:
cd oqs-demos/
git checkout httpd-dilithium3

# Creación del contenedor usando SIG_ALG predeterminado:
docker build -t openquantumsafe/httpd .

# Creación del contenedor cambiando el algoritmo de firma:
docker build --build-arg SIG_ALG=<SIG> \
  -t openquantumsafe/httpd .

# Ejecución del contenedor creado: (TCP 4433)
docker run -p 4433:4433 openquantumsafe/httpd
```

Algoritmo 32: OQS-OpenSSLv1.1.1 y Apache. Ejecución del contenedor.

Según la documentación esta demo hace uso de Dilithium3 como algoritmo de autenticación predeterminado, sin embargo el experimento inicial dio cuenta que el algoritmo utilizado para la creación de las claves fue Dilithium2. Esta inconsistencia se debía a que el `Dockerfile` redefinía por error la variable `SIG_ALG`, variable que permite configurar el algoritmo asimétrico predeterminado al momento de crear el contenedor. Por este motivo se realizó un fork del repositorio oficial, y se corrigió y reportó el error. El Alg. 32 muestra el clonado del repositorio oficial para reproducir los experimentos, y el clonado del repositorio modificado a modo de información. Para

reproducir los experimentos es recomendable utilizar el repositorio oficial ya que el cambio reportado fue integrado a la rama principal del proyecto.

Conexión del cliente OpenSSL.

Las pruebas de conexión se realizaron con la v1.1.1 de OQS-OpenSSL previamente compilada en la Sección VIII.4. Para realizar el siguiente experimento se cargó el módulo de entorno correspondiente, y lanzó el cliente compilado previamente, tal y como se muestra en el Alg. 33.

```
module load openssl/1.1.1
openssl s_client -connect localhost:4433
```

Algoritmo 33: OQS-OpenSSLv1.1.1 y Apache. Ejecución del cliente.

El resultado de la ejecución puede verse en la Fig. 17. Aquí se muestra parte de la salida del comando, donde se ve que se estableció la conexión satisfactoriamente, y marca el uso de Dilithium3 como algoritmo de autenticación y firma digital, el algoritmo que se compiló de manera predeterminada al crear la imagen Docker. Esto implica que ambas implementaciones soportan el algoritmo Dilithium3. Como dato adicional, dicha figura muestra el uso del algoritmo tradicional KEM ECDH con curva Curve25519.

```
subject=CN = oqs-httpd

issuer=CN = oqstest CA

---
No client certificate CA names sent
Peer signature type: Dilithium3
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 9004 bytes and written 483 bytes
Verification error: unable to verify the first certificate
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
```

Figura 17: Conexión OQS-OpenSSL v1.1.1 predeterminada - httpd.

La misma Fig. 17 muetra el error de verificación de certificado digital del servidor. Esto se debe a que el cliente no dispone de la clave pública utilizada para firmar el certificado recibido desde el servidor Apache.

Verificación de certificado digital.

El certificado digital de la CA se encuentra en el directorio `cacert/` y se denomina `CA.crt` (Stebila et al., 2022). Dicho certificado se extrajo utilizando el comando `cat` dentro del contenedor tal y como se muestra en el Alg. 34. Para mantener el orden y consistencia con el resto de los experimentos, se creó un directorio particular dentro de `KEYSDIR` llamado `oqs_httpd/`, y dentro del mismo se extrajo el certificado de la CA y su clave privada por si fuera a ser necesaria en futuros experimentos. Se cargó el módulo de entorno para poder acceder al contenido de la variable `KEYSDIR`.

```
module load openssl/1.1.1
mkdir $KEYSDIR/oqs_httpd/

#Extracción de certificado digital de la CA:
docker run -p 4433:4433 openquantumsafe/httpd \
  cat cacert/CA.crt > $KEYSDIR/oqs_httpd/CA.crt

#Extracción de clave privada de la CA:
docker run -p 4433:4433 openquantumsafe/httpd \
  cat cacert/CA.key > $KEYSDIR/oqs_httpd/CA.key
```

Algoritmo 34: OQS-OpenSSLv1.1.1 y Apache. Extracción del certificado.

El Alg. 35 muestra el comando utilizado para extraer la información del certificado, y un fragmento de interés de su salida (la salida completa se encuentra en el repositorio Git de la tesis (Córdoba, 2022)). Se utilizó la v1.1.1 de OQS-OpenSSL, previamente compilada en este trabajo para extraer dicha información. El certificado digital y su firma son de tipo Dilithium3, algoritmo asimétrico utilizado por defecto esta imagen de Docker.

El comando utilizado para conectar el cliente y verificar dicha firma digital se ve al final del mismo Alg. 35. La Fig. 18 muestra el resultado de la negociación, y puede apreciarse que el certificado ahora pasa el proceso de verificación.

```
# Extracción de datos:
module load openssl/1.1.1
openssl x509 -in $KEYSDIR/oqs_httpd/CA.crt -noout -text

# Salida (resumida para resaltar información de interés):
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: [...]
    Signature Algorithm: dilithium3
    Issuer: CN = oqstest CA
    Validity
            Not Before: Mar 11 14:38:48 2022 GMT
      Not After : Mar 11 14:38:48 2023 GMT
    Subject: CN = oqstest CA
    Subject Public Key Info:
    Public Key Algorithm: dilithium3
      dilithium3 Public-Key:
      pub: [...]
  Signature Algorithm: dilithium3
    [...]

# Verificación de datos del certificado:
openssl s_client -connect localhost:4433 \
  -CAfile $KEYSDIR/oqs_httpd/CA.crt
```

Algoritmo 35: OQS-OpenSSLv1.1.1 y Apache. Extracción de datos.

```
No client certificate CA names sent
Peer signature type: Dilithium3
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 9004 bytes and written 483 bytes
Verification: OK
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 15616 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
```

Figura 18: OQS-OpenSSL v1.1.1 + Apache. Verificación OK.

Este experimento da cuenta de que ambas implementaciones de OpenSSL, la integrada en la imagen Docker de Apache y la compilada para el presente trabajo, soportan algoritmos de clave pública post-cuánticos como Dilithium3. Se agrega como dato adicional que ambas implementaciones de OpenSSL, si bien comparten la versión, v1.1.1, difieren en la letra de patch:

`l` para la release integrada en imagen Docker, y `m` para la compilada para esta tesis. Así, ambas releases ofrecen las mismas características, pero la compilada para el presente trabajo hace correcciones menores respecto de la incluída en la imagen Docker. Para determinar la versión y release del OpenSSL compilado para esta tesis se utilizó el comando `version`. El contenedor Docker no incluye los binarios de OpenSSL, de modo que se extrajo la información analizando el binario de la librería `libssl.so` integrada en el servicio httpd. Ambos procedimientos pueden verse en el Alg. 36.

```
1 # Versión y release de OQS-OpenSSL incluída en Docker
2 docker run -p 4433:4433 openquantumsafe/httpd strings /usr
     /lib/libssl.so.1.1| grep "OpenSSL"
3 > OpenSSL 1.1.1l  24 Aug 2021
4
5 # Versión y release de OQS-OpenSSL compilada manualmente
6 module load openssl/1.1.1 && openssl version
7 > OpenSSL 1.1.1m  14 Dec 2021, Open Quantum Safe 2022-01
```

Algoritmo 36: Versiones de OQS-OpenSSL utilizadas.

Como paso adicional se analizó el certificado del servidor Apache almacenado en el contenedor Docker. El algoritmo utilizado para su generación y firma es Dilithium2, tal y como muestra el Alg. 37 (la salida completa se encuentra en el repositorio Git que acompaña a la tesis (Córdoba, 2022)).

```
# Extracción de certificado digital del servidor:
docker run -p 4433:4433 openquantumsafe/httpd \
  cat /opt/httpd/pki/server.key \
  > $KEYSDIR/oqs_httpd/server.crt

# Extracción de datos:
openssl x509 -in $KEYSDIR/oqs_httpd/server.crt \
  -noout -text

# Salida (resumida para resaltar información de interés):
Certificate: [...]
  Signature Algorithm: dilithium2
  Issuer: CN = oqstest CA
  Validity
    Not Before: Mar  9 14:52:06 2022 GMT
    Not After : Mar  9 14:52:06 2023 GMT
    Public Key Algorithm: dilithium2
    dilithium2 Public-Key: [...]
```

Algoritmo 37: OQS-OpenSSLv1.1.1 y Apache. Certificado servidor.

Intercambio de claves post-cuántico.

El siguiente experimento realizó la incorporación de algoritmos de encapsulamiento de claves KEM. Para ello se llevó a cabo la prueba de conexión utilizando uno de los algoritmos KEM post-cuánticos soportados por OQS-OpenSSL v1.1.1: Frodo640AES. El modificador `-curves` al ejecutar el cliente permite indicar un algoritmo de intercambio de claves (Alg. 38).

```
1 module load openssl/1.1.1
2 openssl s_client -CAfile $KEYSDIR/oqs_httpd/CA.crt \
3   -curves frodo640aes -crlf -connect localhost:4433
```

Algoritmo 38: OQS-OpenSSL v1.1.1. Uso de Frodo640AES.

El resultado de la negociación puede verse en la Fig. 19 e indica que ambas implementaciones pudieron interpretar dicho algoritmo. Esta salida difiere de la salida mostrada en la Fig. 17, en la que se utilizó, de manera predeterminada, el algoritmo de curva elíptica X25519.

```
subject=CN = oqs-httpd

issuer=CN = oqstest CA

---
No client certificate CA names sent
Peer signature type: Dilithium3
Server Temp Key: frodo640aes
---
SSL handshake has read 18740 bytes and written 10027 bytes
Verification: OK
```

Figura 19: OQS-OpenSSL v1.1.1 + Apache. Algoritmo KEM.

Frodo640AES es un algoritmo puramente post-cuántico, como se mencionó en la Sección VII.2.1. En dicha sección también se comentó la integración de algoritmos híbridos junto con los post-cuánticos en el intercambio de claves. Frodo640AES es un algoritmo de nivel de seguridad L1 del NIST (OQS, 2021a) (véase la Sección V.2), por lo que estas implementaciones (servidor y cliente) soportan también el algoritmo híbrido `p256_frodo640aes`, como puede apreciarse en la Fig. 20.

```
No client certificate CA names sent
Peer signature type: Dilithium3
Server Temp Key: p256_frodo640aes_hybrid
---
SSL handshake has read 18805 bytes and written 10092 bytes
Verification: OK
```

Figura 20: OQS-OpenSSL v1.1.1 + Apache. Algoritmo KEM híbrido.

VIII.5.1 Cliente cURL y liboqs

Durante la realización de esta tesis el proyecto OQS publicó una versión de cURL integrada con liboqs. Esta implementación no estaba disponible inicialmente pero resulta de interés comentar un experimento realizado: conectar un contenedor Apache con uno cURL usando cifrado post-cuántico.

Para comunicar el contenedor de `oqs-httpd` y el de `curl` se decidió crear una red Docker. Si bien existen otras alternativas, tales como el uso de la red host provista por Docker, el *mapeo* de puertos del contenedor en puertos del host, o el uso de enlaces (`link`) para que los contenedores puedan ubicarse por nombre, se ha decidió por el uso de una red Docker. Esta red permite comunicar contenedores de manera aislada, sin que se genere tráfico fuera de dicha red virtual, ya sea con otros contenedores externos, o con el propio host que los alberga. La red virtual se creó con el nombre `httpd-test` y luego se lanzó el contenedor Apache, tal y como se ve en el Alg. 39. Este algoritmo muestra también el comando docker que permitió ejecutar el cliente cURL y conectar con el servidor Apache.

```
# Creación de la red docker httpd-test:
docker network create httpd-test

# Ejecución del contenedor Apache:
docker run --network httpd-test --name oqs-httpd -p
   4433:4433 openquantumsafe/httpd

# Ejecución del contenedor cliente cURL (otra terminal):
docker run --network httpd-test -it \
  openquantumsafe/curl curl -k https://oqs-httpd:4433 \
  --curves frodo976aes
```

Algoritmo 39: OQS-OpenSSLv1.1.1 y Apache. Red virtual Docker Apache.

En este caso particular se utilizó un intercambio de claves Frodo640-AES, pero podría omitirse el argumento `--curves` y para utilizar el algoritmo predeterminado `oqs_kem_default` explicado en la Sección VII.2.1.

La negociación TLS requiere que el cliente cURL utilice por defecto el certificado x509 de la CA para verificar la firma del certificado del servidor. Aquí existen dos alternativas: omitir dicha verificación realizando la prueba de conexión con modificador `-k` (o `--insecure`) en `curl`, o descargar el certificado de la CA de Apache y utilizarlo. Dado que en la Sección VIII.5 se extrajo dicho certificado, se optó por esta última opción. El Alg. 40 utiliza la opción `-v` del comando `docker` para crear un volumen dentro del contenedor cURL, mapeando el directorio local donde se encuentra el certificado de la CA, con el directorio `/opt/cacert` de dicho contenedor. Mediante el modificador `--cacert` se le indicó a `curl` la ruta interna donde se encuentra el certificado de la CA. Las opciones `-vvI` se usaron para ampliar la información mostrada por pantalla.

```
docker run --network httpd-test \
  -v $KEYSDIR/oqs_httpd:/opt/cacert -it \
  openquantumsafe/curl curl --cacert /opt/cacert/CA.crt \
  https://oqs-httpd:4433 --curves frodo640aes -vvI
```

Algoritmo 40: OQS-OpenSSLv1.1.1 y Apache. cURL con claves.

Este comando ejecutó el contenedor de cURL, *mapeó* el directorio local al directorio `/opt/cacert` interno de dicho contenedor, y corrió la utilidad `curl`. Se le pasó por argumento el certificado digital situado en el directorio mapeado, y su verificación correcta puede verse en la Fig. 21.

VIII.6 OQS-OPENSSL V1.1.1 Y HTTPS (NGINX)

Uno de los objetivos principales de la presente tesis es el análisis de integraciones post-cuánticas de OpenSSL con Apache para brindar un protocolo HTTPS seguro ante ataques cuánticos. No obstante, durante el desarrollo

```
* successfully set certificate verify locations:
*  CAfile: /opt/cacert/CA.crt
*  CApath: none
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
* TLSv1.3 (IN), TLS handshake, Certificate (11):
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
* TLSv1.3 (IN), TLS handshake, Finished (20):
* TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.3 (OUT), TLS handshake, Finished (20):
* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384
* ALPN, server accepted to use http/1.1
* Server certificate:
*  subject: CN=oqs-httpd
*  start date: Mar 11 14:38:48 2022 GMT
*  expire date: Mar 11 14:38:48 2023 GMT
*  common name: oqs-httpd (matched)
*  issuer: CN=oqstest CA
*  SSL certificate verify ok.
```

Figura 21: Verificación correcta de certificado digital.

del trabajo el proyecto OQS publicó una demo, basada en Docker, de la integración de OQS-OpenSSL v1.1.1 con Nginx v1.16.1. Resulta de especial interés un breve análisis de la misma, dado que Nginx es el servidor web de código abierto más usado después de Apache (Q-Success, 2022).

Al igual que en el caso anterior, como está integrado con OQS-OpenSSL v1.1.1 cuenta con las ventajas (y limitaciones) de esta versión respecto de la v1.0.2. La conexión en esta demo se realizó también en TLS1.3 con mecanismos de autenticación e intercambio de claves post-cuánticos. Igualmente, carece de soporte para cipher suites resistentes, por lo que no pueden realizarse pruebas de concepto en este sentido.

Para evitar ser reiterativos con los comandos, ya que son similares a los utilizados en la Sección VIII.5, se exponen a continuación únicamente los pasos llevados a cabo durante el experimento.

1. Creación de la red virtual Docker para realizar el experimento.
2. Extracción del certificado de la CA del servidor oqs-nginx contenido en la imagen Docker.
3. Ejecución de un servidor oqs-nginx (imagen Docker de OQS).

4. Ejecución del cliente cURL (imagen Docker de OQS) utilizando el certificado de la CA para realizar la verificación de firma en TLS.

La conexión se llevó a cabo satisfactoriamente, y el cliente cURL pudo verificar, mediante el certificado de la CA, la firma del certificado del servidor. Al igual que los experimentos realizados con OQS-OpenSSL y Apache, pudo analizarse el uso de claves y certificados post-cuánticos (Dilithium3 de manera predeterminada) y algoritmos KEM soportados por liboqs.

VIII.7 OQS-OPENSSL V1.1.1 Y OPENVPN

SSL/TLS se utiliza para brindar una capa de seguridad a varios protocolos de aplicación además de HTTP, por ejemplo, SMTP, IMAP, POP, FTP, etc. Entre estos protocolos existen algunos que permiten realizar reds privadas virtuales (VPN, por sus siglas en inglés). Uno de los más utilizado es OpenVPN (OpenVPN Inc., 2018), una herramienta de capa de aplicación multiplataforma y de código abierto.

Microsoft ha realizado un fork de OpenVPN y lo ha integrado con liboqs. Microsoft denominó a este proyecto PQCrypto-VPN (Microsoft, 2018). En este apartado se analizará dicha implementación, y se llevarán a cabo las pruebas de concepto correspondiente.

Se siguieron las guías de compilación e instalación provistas por Microsoft. Estas guías incluyen scripts automatizados que descargan tanto liboqs como OpenVPN, y realizan las compilaciones correspondientes. Luego de compilar e instalar PQCrypto-VPN se analizaron y probaron las configuraciones necesarias para establecer el túnel de red mediante algoritmos post-cuánticos en la autenticación, intercambio de claves y certificados digitales.

Se descargó desde los repositorios la última versión de PQCrypto-VPN liberada por Microsoft (v1.3), en la que la empresa integró OQS-OpenSSL

v1.1.1 con OpenVPN v2.4.9, por lo que se pudieron probar algoritmos KEM y de firma digital post-cuánticos.

A continuación se resumen los experimentos realizados:

1. Compilación y análisis de PQCrypto-VPN.
2. Intercambio de claves tradicional y post-cuántico.
3. Incorporación de algoritmos asimétricos post-cuánticos.
4. Pruebas de conexión y análisis de tráfico.

VIII.7.1 Consideraciones técnicas

Las pruebas de conexión se llevaron a cabo entre dos sistemas GNU/Linux. El servidor OpenVPN es el mismo equipo que se utilizó para realizar las demás pruebas de concepto de OQS explicadas previamente en el presente trabajo. Aprovechando esta ventaja se utilizó OQS-OpenSSL v1.1.1 para generar certificados digitales basados en algoritmos post-cuánticos. El equipo cliente utilizado fue un Arch Linux, pero este detalle es intrascendente, tal y como se comentó en la Sección VIII, ya que OpenVPN actúa tanto de cliente como de servidor dependiendo de las configuraciones que se le suministren por línea de comandos, o mediante archivos de configuración. La ejecución de OpenVPN se llevó a cabo utilizando un módulo de entorno para evitar conflictos con librerías criptográficas del sistema operativo.

Para identificar ambos equipos se asignó la dirección IP `192.168.0.20` al servidor y la IP `192.168.0.10` al cliente. Esta última es despreciable para las pruebas debido a que no se requiere en ninguna configuración, no obstante es de interés en el análisis de capturas de tráfico y registros de conexión.

El primer paso fue clonar el repositorio oficial de PQCrypto-VPN. El repositorio Git contiene un directorio llamado `openvpn/`, que a su vez incluye dos directorios de interés: `build/` y `config/`. El directorio `build/` contiene

las herramientas necesarias para compilar y poner en marcha el OpenVPN, mientras que `config/` provee archivos de configuración de ejemplo.

Una vez clonado el repositorio se utilizó el script `openvpn/build/build.py` para realizar la compilación de la aplicación. Los comandos mostrados en el Alg. 41 permitieron crear el directorio de trabajo, clonar el repositorio de PQCrypto-VPN, y compilarlo utilizando el mencionado script.

```
# Clonado del repositorio:
mkdir /opt/openvpn_ms/PQCryptoVPN_1.3/
cd /opt/openvpn_ms/PQCryptoVPN_1.3/
git clone --recurse-submodules https://github.com/
    microsoft/PQCrypto-VPN.git

# Compilación de PQCryptoVPN:
cd PQCrypto-VPN_1.3/openvpn/build/
./build.py
```

Algoritmo 41: PQCryptoVPN. Clonado del repositorio.

Los primeros intentos de compilación generaron errores al momento de enlazar los binarios de test incorporados con liboqs. Dado que el presente trabajo no se centra en ejecutar dichas pruebas, se omitió su compilación y enlace activando la macro `OQS_BUILD_ONLY_LIB` en el archivo `CMakeLists.txt` de liboqs provisto por Microsoft. Este archivo incluye las configuraciones de compilación y enlace que utilizará la utilidad `ninja` para compilar. Para activar la macro mencionada se cambió su valor de `OFF` a `ON`, (Alg. 42).

```
39 #Archivo:
40 # /opt/openvpn_ms/PQCryptoVPN_1.3/openvpn/build/stage/usr/
       local/openvpn/build/repos/liboqs/CMakeLists.txt
41
42 option(OQS_BUILD_ONLY_LIB "Build only liboqs and do not
       expose build targets for tests, documentation, and
       pretty-printing available." ON)
```

Algoritmo 42: PQCryptoVPN. Clonado del repositorio.

La compilación y enlace, luego de este cambio, se llevaron a cabo satisfactoriamente, y se crearon los siguientes elementos de interés, que serán de utilidad para las pruebas de concepto:

- Directorio `stage/`: productos compilados (OpenVPN y OQS-OpenSSL).
- Archivo `pq-openvpn-linux-staged.tar.gz`: tarball que empaqueta el directorio stage/ para distribuir la aplicación.
- Archivo `openvpn-install-2.4.9-I601-Win10.exe`: Compilación de OpenVPN integrado con OQS-OpenSSL para poder se instalado en sistemas Windows.
- Directorio `docker/`: contiene el Dockerfile necesario para realizar la construcción de PQCrypto-VPN en un contenedor Docker.

Dentro del directorio `stage/` se encuentra `stage/usr/local/openvpn/`, un subdirectorio que contiene los siguientes elementos:

- `lib/`: bibliotecas compiladas de OQS-OpenSSL v1.1.1, a saber, `libcrypto.so.1.1` y `libssl.so.1.1`, necesarias para OpenVPN con soporte de algoritmos post-cuánticos.
- `sbin/`: Contiene el binario ejecutable de OpenVPN.
- `etc/`: En este directorio (vacío inicialmente) se cargarán las configuraciones para las pruebas.

Con esta información se creó el módulo de entorno que se ve en el Alg. 43.

```
#%Module1.0#####################################################
##
## modules modulefile
##
## File: /usr/share/modules/modulefiles/pqcryptovpn/1.3

set prefix "PQCrypto-VPN"
set version "1.3"

set TOPDIR      /opt
set BUILDDIR  /opt/openvpn_ms/PQCryptoVPN_1.3
set BASEDIR $BUILDDIR/openvpn/build/stage/usr/local/
   openvpn
setenv BUILDDIR $BUILDDIR
setenv KEYSDIR  $TOPDIR/keys
setenv LD_LIBRARY_PATH $BASEDIR/lib
setenv BASEDIR $BASEDIR
setenv CONFDIR $BASEDIR/etc/config

prepend-path  PATH $BASEDIR/sbin
```

Algoritmo 43: PQCryptoVPN v1.3. Módulo de entorno.

Se verificó que el módulo estuviera cargue la versión correcta de OpenVPN como se ve en el Alg. 44. Esta versión de PQCrypto-VPN, al basar su seguridad en OQS-OpenSSL v1.1.1, no incorpora cipher suites post-cuánticas, por lo que los comandos de consulta de cifradores no arrojan diferencias respecto de una compilación tradicional de OpenVPN.

```
# Módulo de entorno:
module load pqcryptovpn/1.3

# Verificación de versión:
openvpn --version
>
OpenVPN 2.4.9 x86_64-pc-linux-gnu [SSL (OpenSSL)] [LZO] [
   LZ4] [EPOLL] [MH/PKTINFO] [AEAD] built on Mar 20 2022
library versions: OpenSSL 1.1.1g  21 Apr 2020, Open
   Quantum Safe 2020-07 snapshot, LZO 2.10
Originally developed by James Yonan
[...]
```

Algoritmo 44: PQCryptoVPN v1.3. Versión.

Para las pruebas se usaron las claves y certificados provistos por la suite, situados en `openvpn/build/repos/openvpn-2.4.9/sample/sample-keys/`. Este directorio se copió dentro de `/opt/keys/`, que ya se disponía de las implementaciones anteriores, con el objetivo de facilitar el acceso en los experimentos y mantener centralizadas las claves y certificados utilizados. Con la misma intención se copió el directorio `openvpn/config` en `etc/`. Estos comandos pueden verse en el Alg. 45.

```
#Módulo de entorno:
module load pqcryptovpn/1.3

# Copia de claves:
cp -r $BUILDDIR/openvpn/build/repos/openvpn-2.4.9/sample/
   sample-keys $KEYSDIR

# Copia de archios de configuración:
cp -r $BUILDDIR/openvpn/config $CONFDIR
```

Algoritmo 45: PQCryptoVPN. Copiado de archivos.

Dado que la implementación es la misma tanto para el servidor como para el cliente, para realizar las pruebas de conexión se replicaron el directorio

de la implementación, `openvpn/`, y el de claves y certificados, `/opt/keys/`, desde el equipo servidor al sistema cliente. De esta forma fue posible ejecutar PQCrypto-VPN en modo servidor en uno de los equipos, y en modo cliente en el otro.

VIII.7.2 Intercambio de claves post-cuántico

En el equipo servidor se cargaron las configuraciones necesarias para hacer uso de claves, certificados y módulos DH predeterminados del proyecto, ahora ubicados en el directorio `$KEYSDIR`. Un trabajo similar se llevó a cabo en el equipo cliente. Ambas configuraciones pueden verse en el Alg. 46.

```
# Dentro de (/opt/openvpn_ms/PQCryptoVPN_1.3/openvpn/build
   /stage/usr/local/openvpn/)

# Archivo ./etc/config/server.ovpn
ca /opt/keys/sample-keys/ca.crt
cert /opt/keys/sample-keys/server.crt
key /opt/keys/sample-keys/server.key
dh /opt/keys/sample-keys/dh2048.pem

# Archivo ./etc/config/client.ovpn
remote 192.168.0.20 1194     # dirección IP del servidor
ca /opt/keys/sample-keys/ca.crt
cert /opt/keys/sample-keys/client.crt
key /opt/keys/sample-keys/client.key
```

Algoritmo 46: PQCryptoVPN. Configuraciones.

Una vez almacenadas las configuraciones en los archivos `server.ovpn` y `client.ovpn` se ejecutaron el servidor y el cliente OpenVPN respectivamente en los sistemas servidor y cliente como se ve en el Alg. 47.

```
# Ejecución Servidor:
module load pqcryptovpn/1.3
openvpn --config $CONFDIR/server.ovpn

# Ejecución Cliente:
module load pqcryptovpn/1.3
openvpn --config $CONFDIR/client.ovpn
```

Algoritmo 47: PQCryptoVPN. Ejecución.

El registro de conexión del cliente no hace mención a mecanismos post-cuánticos, mientras que el servidor muestra explícitamente el uso de algoritmos tradicionales, tal y como se aprecia en la Fig. 22.

```
Sun Mar 20 17:05:28 2022 us=648161 192.168.0.10:51939 UDPv4 READ [22] from [AF_INET]192.168.
0.10:51939: P_ACK_V1 kid=0 [ 6 ]
Sun Mar 20 17:05:28 2022 us=648363 192.168.0.10:51939 Control Channel: TLSv1.3, cipher TLSv1
.3 TLS_AES_256_GCM_SHA384, group_id 1034 (NOT post-quantum key exchange), 2048 bit RSA
Sun Mar 20 17:05:28 2022 us=648629 192.168.0.10:51939 [Test-Client] Peer Connection Initiate
d with [AF_INET]192.168.0.10:51939
Sun Mar 20 17:05:28 2022 us=648842 Test-Client/192.168.0.10:51939 MULTI_sva: pool returned I
Pv4=10.8.0.6, IPv6=(Not enabled)
```

Figura 22: PQcryptoVPN. Uso de intercambio tradicional.

Se habilitó la opción `ecdh-curve sikep434` en las configuraciones de servidor y cliente. Esta opción permite especificar una curva elíptica para ser utilizada en el intercambio DH. Se volvió a establecer la conexión y ahora los nodos utilizaron el intercambio de claves basado en el algoritmo post-cuántico Sike-p434. Esto se llevó a cabo aún utilizando las mismas claves y certificados tradicionales basados en RSA. El resultado del intercambio de claves post-cuántico usando Sike-p434 puede verse en las Figs. 23 (servidor) y 24 (cliente). Si bien Sike-p434 es el algoritmo de intercambio de claves configurado en este experimento, puede utilizarse cualquiera de los mencionados al inicio de este capítulo (véase Sección VII.2.1). Por ejemplo, la Fig. 25 muestra una captura del cliente cuando ambos nodos negociaron un intercambio basado en Frodo640-AES.

```
Sun Mar 20 17:11:12 2022 us=576614 192.168.0.10:47008 UDPv4 READ [22] from [AF_INET]192.168.
0.10:47008: P_ACK_V1 kid=0 [ 5 ]
Sun Mar 20 17:11:12 2022 us=577509 192.168.0.10:47008 UDPv4 READ [22] from [AF_INET]192.168.
0.10:47008: P_ACK_V1 kid=0 [ 6 ]
Sun Mar 20 17:11:12 2022 us=577695 192.168.0.10:47008 Control Channel: TLSv1.3, cipher TLSv1
.3 TLS_AES_256_GCM_SHA384, group_id 1287 (post-quantum key exchange), 2048 bit RSA
Sun Mar 20 17:11:12 2022 us=577922 192.168.0.10:47008 [Test-Client] Peer Connection Initiate
d with [AF_INET]192.168.0.10:47008
Sun Mar 20 17:11:12 2022 us=577952 Test-Client/192.168.0.10:47008 MULTI_sva: pool returned I
Pv4=10.8.0.6, IPv6=(Not enabled)
```

Figura 23: PQcryptoVPN. Intercambio post-cuántico - Servidor.

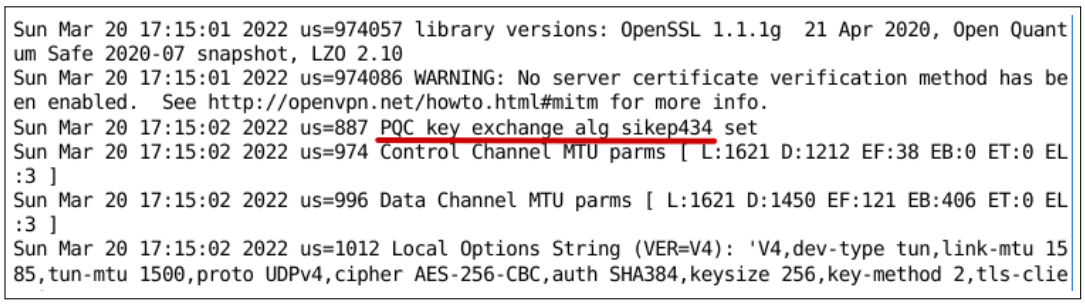

Figura 24: PQcryptoVPN. Intercambio post-cuántico - Cliente.

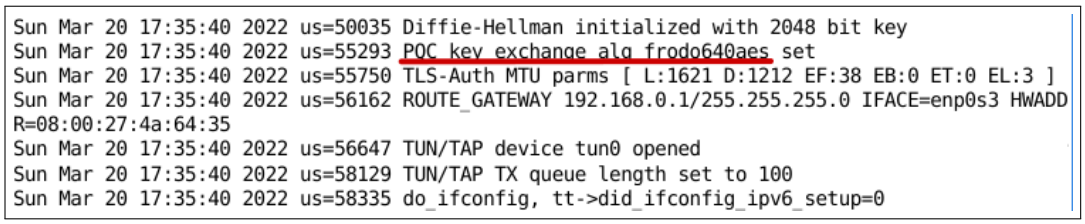

Figura 25: PQcryptoVPN. Intercambio frodo640aes - Cliente.

VIII.7.3 Incorporación de criptografía post-cuántica

En esta sección se analiza la conexión OpenVPN haciendo uso de claves y certificados digitales basados en algoritmos asimétricos post-cuánticos en lugar del tradicional RSA. El repositorio de PQCrypto-VPN no incluye archivos de claves y certificados post-cuánticos como ejemplo, por lo que se debieron generar manualmente para montar las pruebas de concepto. Si bien las claves y certificados podrían haberse generado utilizando la implementación de OQS-OpenSSL v1.1.1 compilada en la sección VIII.4, se decidió utilizar la implementación de OpenSSL provista por Microsoft en PQCrypto-VPN para facilitar la reproducción de los experimentos sin generar dependencias externas.

El primer paso fue crear un módulo de entorno que permitiera ejecutar el OQS-OpenSSL v1.1.1 incluido por Microsoft (Alg. 48). Este módulo de entorno, además, facilitó la comparación de la versión de release del OQS-OpenSSL v1.1.1 compilada previamente en este trabajo, con la incluida por Microsoft en PQCryto-VPN. Esta información puede verse en el Alg. 49.

```
#%Module1.0#####################################################
##
## modules modulefile
##
## File: /usr/share/modules/modulefiles/openssl/1.1.1_ms

set prefix "OQS-OpenSSL PQCrypto-VPN"
set version "1.1.1"

set TOPDIR      /opt
set BASEDIR  /opt/openvpn_ms/PQCryptoVPN_1.3/openvpn/build
   /repos/openssl-oqs
setenv KEYSDIR  $TOPDIR/keys
setenv APPSDIR $BASEDIR/apps
setenv LD_LIBRARY_PATH $BASEDIR

prepend-path  PATH $BASEDIR/apps
```

Algoritmo 48: PQCryptoVPN v1.3. Módulo de entorno de OQS-OpenSSL.

Puede apreciarse que la versión provista por Microsoft data de Abril del 2020, fecha en la que liberó el repositorio de PQCrypto-VPN, y no se ha actualizado desde entonces.

```
module load openssl/1.1.1
openssl version
>
OpenSSL 1.1.1m  14 Dec 2021, Open Quantum Safe 2022-01

module load openssl/1.1.1_ms
openssl version
>
OpenSSL 1.1.1g  21 Apr 2020, Open Quantum Safe 2020-07
```

Algoritmo 49: PQCryptoVPN v1.3. .

Para llevar a cabo los experimentos de conexión utilizando algoritmos asimétricos post-cuánticos se generaron los siguientes elementos:

- Clave privada y certificado de la CA.
- Clave privada y certificado digital del servidor firmado por la CA.
- Clave privada y certificado digital del cliente firmado por la CA.

Estos elementos se generaron y firmaron en el equipo servidor, y luego se replicaron al cliente únicamente aquellos que necesarios para completar su configuración, a saber:

- Clave privada del cliente.
- Certificado digital del cliente.
- Certificado digital de la CA para verificar las firmas.

En el presente trabajo se realizaron pruebas de conexión utilizando los algoritmos QTeslaP-I (`qteslapi`), QTeslaP-III (`qteslapiii`), Dilithium4 con curva P384 (`p384_dilithium4`) y QTeslaP-III con curva P384 (`p384_qteslapiii`). Con todos se llevó a cabo la autenticación correctamente. A fines de documentación se detallará la conexión utilizando `p384_qteslapiii`, elegido de manera arbitraria para este caso. El experimento consistió en:

1. Generación de certificado digital y clave privada de la CA basados en `p384_qteslapiii`.
2. Generación de certificado digital y clave privada del servidor, y firma del certificado mediante la clave privada de la CA.
3. Generación de certificado digital y clave privada del cliente, y firma del certificado mediante la clave privada de la CA.
4. Copiado de archivos al equipo cliente: certificado digital y clave del cliente, y certificado digital de la CA.
5. Actualización de los archivos de configuración de servidor y cliente para incorporar los elementos generados.
6. Ejecución del servidor y del cliente para establecer la conexión.

Generación de certificados y claves

Primero se generó el certificado digital y la clave privada de la autoridad certificante utilizando `p384_qteslapiii`. Las claves resultantes de crearon en el directorio `/opt/keys/`. Segundo, se generó la clave privada y la solicitud de firma de certificado para el servidor OpenVPN. Se utilizó el mismo algoritmo en este experimento. Luego se firmó dicho certificado utilizando

la clave privada de la CA. El mismo procedimiento se utilizó para generar la clave y certificado del cliente. El Alg. 50 muestra el detalle de los comandos.

```
# Carga del módulo de entorno:
module load openssl/1.1.1_ms

# Generación de clave y certificado dse la CA:
openssl req -x509 -new -newkey p384_qteslapiii \
    -keyout $KEYSDIR/p384_qteslapiii_CA.key \
    -out $KEYSDIR/p384_qteslapiii_CA.crt \
    -nodes -subj "/CN=CA p384_qteslapiii" -days 365 \
    -config $APPSDIR/openssl.cnf

# Generación de clave y solicitud de firma del servidor:
openssl req -new -newkey p384_qteslapiii \
  -keyout $KEYSDIR/p384_qteslapiii_srv.key \
  -out $KEYSDIR/p384_qteslapiii_srv.csr \
  -nodes -subj "/CN=SRV p384_qteslapiii" -days 365 \
  -config $APPSDIR/openssl.cnf

# Firma del certificado del servidor:
openssl x509 -req -in $KEYSDIR/p384_qteslapiii_srv.csr \
  -out $KEYSDIR/p384_qteslapiii_srv.crt \
  -CA $KEYSDIR/p384_qteslapiii_CA.crt \
  -CAkey $KEYSDIR/p384_qteslapiii_CA.key \
  -CAcreateserial -days 365

# Generación de clave y solicitud de firma cliente:
openssl req -new -newkey p384_qteslapiii \
  -keyout $KEYSDIR/p384_qteslapiii_cli.key \
  -out $KEYSDIR/p384_qteslapiii_cli.csr \
  -nodes -subj "/CN=CLI p384_qteslapiii" -days 365 \
  -config $APPSDIR/openssl.cnf

# Firma del certificado del cliente:
openssl x509 -req -in $KEYSDIR/p384_qteslapiii_cli.csr \
  -out $KEYSDIR/p384_qteslapiii_cli.crt \
  -CA $KEYSDIR/p384_qteslapiii_CA.crt \
  -CAkey $KEYSDIR/p384_qteslapiii_CA.key \
  -CAcreateserial -days 365
```

Algoritmo 50: PQCryptoVPN. Generación y firma de claves.

Copia de archivos, configuraciones y ejecución.

Se copiaron los archivos necesarios al equipo cliente: clave y certificado del cliente, y certificado de la autoridad certificante. Para mantener el orden, dado que los elementos mencionados se crearon en el directorio `/opt/keys` (`$KEYSDIR`) del servidor, se copiaron el la misma ubicación en el cliente.

Luego se actualizaron los archivos de configuración para referenciar los nuevos elementos generados. El Alg. 51 muestra la configuración final de servidor y cliente, y los comandos necesarios para su ejecución.

```
# Archivo $CONFDIR/server-pq.ovpn
ca /opt/keys/p384_qteslapiii_CA.crt
cert /opt/keys/p384_qteslapiii_srv.crt
key /opt/keys/p384_qteslapiii_srv.key
dh /opt/keys/sample-keys/dh2048.pem

# Archivo $CONFDIR/client-pq.ovpn
remote 192.168.0.20 1194     # dirección IP del servidor
ca /opt/keys/p384_qteslapiii_CA.crt
cert /opt/keys/p384_qteslapiii_cli.crt
key /opt/keys/p384_qteslapiii_cli.key

# Ejecución del servidor:
module load pqcryptovpn/1.3
openvpn --config $CONFDIR/server-pq.ovpn

# Ejecución del cliente:
module load pqcryptovpn/1.3
openvpn --config $CONFDIR/client-pq.ovpn
```

Algoritmo 51: PQCryptoVPN. Configuraciones.

Si bien en las salidas estos comandos de conexión del OpenVPN no muestra los algoritmos que dieron origen a certificados digitales y claves, la conexión se llevó a cabo satisfactoriamente, y ambos nodos pudieron verificar a su contraparte, como puede observarse en los extractos de la salida mostrados en el Alg. 52.

```
# Salida del Servidor verifica cert. de CA y cliente:
[...]
Sun Mar 20 19:23:27 2022 us=22393 192.168.0.10:47803
   VERIFY OK: depth=1, CN=CA p384_qteslapiii
Sun Mar 20 19:23:27 2022 us=25295 192.168.0.10:47803
   VERIFY OK: depth=0, CN=CLI p384_qteslapiii
[...]

# Salida del Cliente verifica cert. de CA y servidor:
[...]
Sun Mar 20 19:23:26 2022 us=944309 VERIFY OK: depth=1, CN=
   CA p384_qteslapiii
Sun Mar 20 19:23:26 2022 us=947149 VERIFY OK: depth=0, CN=
   SRV p384_qteslapiii
[...]
```

Algoritmo 52: PQCryptoVPN. Verificación de firmas.

Análisis de captura de tráfico.

Al establecer la conexión TLS el cliente ofrece los algoritmos de firma digital soportados. Un fragmento del tráfico capturado se ve en la Fig. 26.

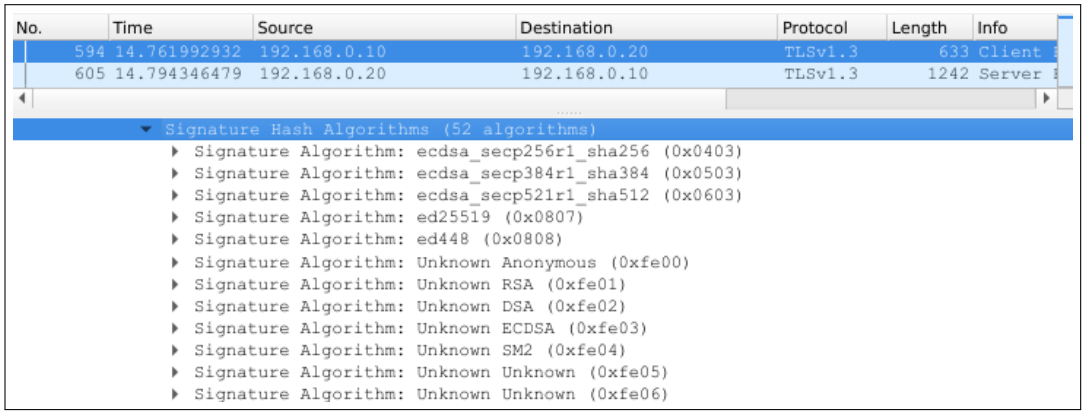

Figura 26: Algoritmos de firma digital PQCrypto-VPN.

En este caso particular se muestran menos algoritmos de firma que los utilizados por OQS-OpenSSL v1.1.1 nativo mostrados en la Fig. 13. Esto se debe a que, si bien las versiones de OQS-OpenSSL (v1.1.1) y de la librería liboqs (v0.3.0) analizadas con anterioridad en esta tesis son las mismas que las utilizadas en PQCrypto-VPN, esta última data de Abril de 2020, mientras que la rama estable analizada en este trabajo es de Diciembre de 2021.

Uno de los cambios más visibles es el número de identificación de cada algoritmo de firma digital. Por ejemplo, el algoritmo Dilithium3 incluido en PQCrypto-VPN se identifica con el código `0xfe06`, y en la versión de OQS-OpenSSL compilada previamente para esta tesis tiene el código `0xfea3`. Además esta última versión incorpora algoritmos no disponibles en la implementación de Microsoft, como Dilithium5 y sus variantes híbridas.

El archivo de cabecera `ssl_local.h`, ubicado dentro del directorio `ssl` de la implementación de OQS-OpenSSL incluye las macros que definen los algoritmos mostrados en la Fig. 13. Particularmente el archivo es el siguiente:

```
$BUILDDIR/openvpn/build/repos/openssl-oqs/ssl/ssl_local.h
```

Para analizar las diferencias entre los algoritmos soportados por el OQS-OpenSSL v1.1.1 compilado en el presente trabajo y los disponibles en la release de PQCrypto-VPN, se creó un archivo de diferencias, que puede encontrarse en el repositorio Git que acompaña a esta tesis (Córdoba, 2022).

```
module load openssl/1.1.1_ms
openssl list -public-key-algorithms | grep Name \
  | sort > openssl_1.1.1_publickey_algos_sortnames.txt

module load openssl/1.1.1
openssl list -public-key-algorithms | grep Name \
  |sort > openssl_1.1.1_publickey_algos_sortnames.txt

diff openssl_1.1.1ms_publickey_algos_sortnames.txt \
  openssl_1.1.1_publickey_algos_sortnames.txt \
  > openssl_1.1.1ms_vs_1.1.1_publickey_algos.diff
```

Algoritmo 53: PQCryptoVPN. Listar algoritmos.

Ambas implementaciones coinciden en los algoritmos de clave pública tradicionales, pero difieren en los algoritmos asimétricos post-cuánticos soportados. Mayormente las diferencias se deben a que la última versión de OQS-OpenSSL quitó el soporte a muchos algoritmos que no quedaron finalistas en la 3ra ronda de estandarización del NIST. A continuación se listan algunas diferencias importantes de la versión de OQS-OpenSSL utilizada en el presente trabajo, respecto de la provista por Microsoft.

- Se eliminó el soporte para Dilithium4 y su variante híbrida p384.
- Se eliminó el soporte para mqdss3148 y su variante híbrida p256.
- El algoritmo p256_dilithium3 cambió a la curva p384.
- Se quitó soporte para el algoritmo Picnic2 puro y variantes híbridas.
- Se eliminó el soporte para QteslapIII y su variante híbrida p256.
- Se agregó el algoritmo Dilithium5, y variantes de Dilithium 2, 3 y 5 combinadas con AES.
- Se agregó el algoritmo Picnic3 y su variante híbrida con curva p256.

IX EXPERIMENTACIÓN: WOLFSSL

A continuación se detalla la realización de una prueba de concepto de conexión post-cuántica basada en NTRU utilizando wolfSSL, y su posterior integración con el servicio HTTPS Apache.

Los experimentos realizados en este capítulo se resumen a continuación:

1. Compilación, análisis y prueba de wolfSSL.
2. Análisis y prueba de QSH en wolfSSL.
3. Análisis y prueba de NTRU en wolfSSL.
4. Integración, compilación y análisis de wolfSSL+NTRU con Apache.
5. Integración y compilación de wolfSSL con OpenVPN.

Nota de obsolescencia

Debe aclararse que los experimentos realizados a continuación se llevaron a cabo utilizando la versión 4.3.0 de wolfSSL, última disponible en su momento con soporte para NTRU. En Noviembre de 2021 wolfSSL liberó la versión 5.0.0 de su implementación, y entre los cambios relacionados con criptografía post-cuántica se encuentran los siguientes (WolfSSL, Inc., 2021b):

- Integración con liboqs v0.7.0 con los algoritmos KEMs finalistas de la 3ra ronda de estandarización del NIST. Este soporte no estaba disponible al momento de realizar los experimentos.
- Integración de algoritmos híbridos basados en OQS y curvas NIST.
- Eliminación del soporte para algoritmos post-cuanticos NTRU y QSH.

El último punto indica que, con las versiones de wolfSSL >5.0.0 no será posible reproducir los experimentos detallados a continuación. El capítulo

decidió mantenerse igualmente dado que permite estudiar las negociaciones QSH/NTRU en un intercambio TLS, y NTRU aún se mantiene como uno de los algoritmos finalistas del proceso de estandarización que poseen mejor rendimiento.

Por otro lado, el presente capítulo concluye con la incorporación de criptografía post-cuántica NTRU en Apache para lograr un servicio HTTPS post-cuántico. Dicha integración fue posible gracias a la incorporación de QSH/NTRU en wolfSSL como se detalla a continuación.

IX.1 CONSIDERACIONES TÉCNICAS

Los experimentos realizados con wolfSSL se llevaron a cabo en el mismo equipo virtual Lubuntu utilizado para las pruebas de OQS, y previamente descrito en la Sección VIII.1. Se utilizaron tres directorios base:

- `/opt/wolfssl` para wolfSSL y su integración con NTRU y con Apache.
- `/opt/wolfssl_openvpn` para la implementación de Openvpn integrada con wolfSSL.**(*)**
- `/opt/keys` para la centralización de claves privadas, públicas y certificados digitales x509 utilizados durante las pruebas **(*)**.

Los experimentos de integración entre wolfSSL y NTRU con Apache se realizaron en el sistema virtual mencionado, mientras que los experimentos de VPN requirieron un segundo sistema que oficiara de cliente. Se utilizó el mismo sistema cliente descripto en la Sección VIII.1. Los directorios marcados con (*) representan los elementos copiados a este sistema cliente.

IX.2 INTEGRACIÓN WOLFSSL+QSH/NTRU

IX.2.1 Compilación e instalación

Para compilar wolfSSL con NTRU primero instaló la biblioteca NTRU en el sistema operativo. A tal propósito se creó el directorio `/opt/wolfssl/`,

dentro del mismo se clonó la librería NTRUEncrypt, se compiló e instaló en el sistema. Estos pasos se muestran en el Alg. 54.

```
# Clonado de NTRUEncrypt:
mkdir /opt/wolfssl/ && cd /opt/wolfssl/
sudo git clone \
  https://github.com/NTRUOpenSourceProject/NTRUEncrypt.git

# Compilación e instalación de NTRUEncrypt:
cd NTRUEncrypt/
./autogen.sh
./configure
make && make install
```

Algoritmo 54: wolfSSL. Instalación de la librería NTRUEncrypt.

De manera predeterminada la librería NTRUEncrypt se instaló en el directorio `/usr/local/lib/` tal y como puede verse en la Fig. 27.

```
root@juncotic-lubuntu:/opt/wolfssl# ls -l /usr/local/lib/ --color=no
total 3536
-rw-r--r-- 1 root root   352930 feb 11 11:32 libntruencrypt.a
-rwxr-xr-x 1 root root      981 feb 11 11:32 libntruencrypt.la
lrwxrwxrwx 1 root root       23 feb 11 11:32 libntruencrypt.so -> libntruencrypt.so.0.1.0
lrwxrwxrwx 1 root root       23 feb 11 11:32 libntruencrypt.so.0 -> libntruencrypt.so.0.1.0
-rwxr-xr-x 1 root root   233120 feb 11 11:32 libntruencrypt.so.0.1.0
-rwxr-xr-x 1 root root      956 feb 26 19:36 libwolfssl.la
lrwxrwxrwx 1 root root       20 feb 26 19:36 libwolfssl.so -> libwolfssl.so.24.0.0
lrwxrwxrwx 1 root root       20 feb  4 16:57 libwolfssl.so.23 -> libwolfssl.so.23.0.0
-rwxr-xr-x 1 root root  1486376 feb  4 16:57 libwolfssl.so.23.0.0
lrwxrwxrwx 1 root root       20 feb 26 19:36 libwolfssl.so.24 -> libwolfssl.so.24.0.0
-rwxr-xr-x 1 root root  1522024 feb 26 19:36 libwolfssl.so.24.0.0
```

Figura 27: Librería NTRU.

El siguiente paso fue compilar wolfSSL. Si bien al momento de realizar esta prueba se encuentra disponible la versión `v4.4.0-stable`, la misma no se encuentra adaptada para su integración con NTRU, y produce errores de compilación. A raíz de esto se optó por utilizar la versión `v4.3.0-stable` para poder llevar a cabo las pruebas. Como puede observarse en el Alg. 55, el script `./configure` recibió por argumento algunas opciones de interés:

- `--enable-qsh`: Habilita QSH (véase Sección III.2).
- `--with-ntru=...`: Directorio donde se instaló NTRUEncrypt.
- `--enable-apachehttpd`: Activa la integración con Apache (Sección IX.3).

```
# Clonado de wolfSSL:
cd /opt/wolfssl/
mkdir wolfssl-git/ && cd wolfssl-git
git clone https://github.com/wolfSSL/wolfssl.git

# Cambio de rama a v4.3.0-stable (compatible con NTRU):
cd wolfssl-git/ && git checkout v4.3.0-stable

# Compilación e instalación de wolfSSL:
./autogen.sh
./configure --enable-qsh --with-ntru=/usr/local/lib \
  -enable-opensslextra --enable-supportedcurves \
  --enable-apachehttpd --enable-tlsx --enable-all \
  --enable-ecc --enable-psk --enable-aesccm \
  C_EXTRA_FLAGS="-DWOLFSSL_STATIC_RSA -DHAVE_SNI"
make && make install
```

Algoritmo 55: wolfSSL. Instalación.

Debe notarse también que se habilitó un flag especial `WOLFSSL_STATIC_RSA` para forzar a wolfSSL a utilizar claves RSA estáticas en lugar de efímeras. Si bien esto es desaconsejado en servicios de producción, existen sistemas criptográficos como NTRU que requieren de claves estáticas para funcionar (WolfSSL, 2021, 2020a).

IX.2.2 Negociación SSL/TLS

wolfSSL incluye un servidor y un cliente de ejemplo para realizar la verificación de conexión y funcionamiento. Además, como se trata de una implementación de código abierto, los fuentes de estos ejemplos sirven para orientar el desarrollo de aplicaciones que hagan uso de SSL/TLS. La compilación de wolfSSL detallada en el apartado anterior generó los binarios ejecutables de servidor y cliente. Para ejecutar servidor y cliente de ejemplo se usaron los comandos mostrados en el Alg. 56.

```
# En el equipo servidor:
cd /opt/wolfssl/wolfssl-git/ && ./examples/server/server
# En el equipo cliente:
cd /opt/wolfssl/wolfssl-git/ && ./examples/client/client
```

Algoritmo 56: wolfSSL. Ejecución de servidor y cliente.

El servidor por defecto atiende en el puerto TCP 11111. La salida estándar del mismo, sin ninguna configuración adicional, se puede ver en la Fig. 28.

```
root@juncotic-lubuntu:/opt/wolfssl/wolfssl-git# ./examples/server/server
peer's cert info:
 issuer : /C=US/ST=Montana/L=Bozeman/O=wolfSSL_2048/OU=Programming-2048/CN=www.wolfs
sl.com/emailAddress=info@wolfssl.com
 subject: /C=US/ST=Montana/L=Bozeman/O=wolfSSL_2048/OU=Programming-2048/CN=www.wolfs
sl.com/emailAddress=info@wolfssl.com
 serial number:aa:c4:bf:4c:50:bd:55:77
SSL version is TLSv1.2
SSL cipher suite is QSH:TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL curve name is FFDHE_3072
Server Random : 18CE52B5754C26DD14ED3B707F769EAF99988F9DB5689A5D9364B5D39CBF7E09
Client message: hello wolfssl!
root@juncotic-lubuntu:/opt/wolfssl/wolfssl-git#
```

Figura 28: Respuesta servidor wolfSSL negociación estándar.

La interacción del cliente puede observarse en la Fig. 29. Como puede apreciarse en ambas capturas, la negociación SSL/TLS estándar utiliza TLS1.2, la cipher suite `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`, y usa QSH.

```
root@juncotic-lubuntu:/opt/wolfssl/wolfssl-git# ./examples/client/client
Session Ticket CB: ticketSz = 142, ctx = initial session
peer's cert info:
 issuer : /C=US/ST=Montana/L=Bozeman/O=Sawtooth/OU=Consulting/CN=www.wolfssl.com/ema
ilAddress=info@wolfssl.com
 subject: /C=US/ST=Montana/L=Bozeman/O=wolfSSL/OU=Support/CN=www.wolfssl.com/emailAd
dress=info@wolfssl.com
 serial number:01
SSL version is TLSv1.2
SSL cipher suite is QSH:TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL curve name is SECP256R1
Session timeout set to 300 seconds
Client Random : B130C9062F68C9993DE18F7E4D53EFAB18BE8E49D01A5757CFD21B3B5BECCD21
```

Figura 29: Respuesta cliente wolfSSL negociación estándar.

El cliente provee algunos comandos y opciones adicionales que pueden listarse con la opción `-h`. Entre ellos se encuentra el modificador `-e`, que permite listar las cipher suites soportadas. Como se ha realizado la compilación utilizando los flags `NTRU` y `QSH`, estos algoritmos se verán incluidos en la salida, así como también los algoritmos mencionados al principio de esta sección. Para listar las cipher suites de forma más usó `sed` para editar la salida del comando, reemplazando el caracter `":"` por un salto de línea:

```
cd /opt/wolfssl/wolfssl-git
./examples/client/client -e|sed 's/:/\n/g'
```

Este comando lista todas las cipher suites soportadas por wolfSSL. Como el interés de este trabajo se centra en criptografía post-cuántica, a continuación se muestra un listado de las cipher suites que son de mayor importancia. El listado completo puede encontrarse en el repositorio Git que acompaña a esta tesis (Córdoba, 2022).

- NTRU-RC4-SHA
- NTRU-DES-CBC3-SHA
- NTRU-AES128-SHA
- NTRU-AES256-SHA
- QSH

La opción `-l` del cliente permite configurar una cipher suite específica, o varias separadas por `":"`. La negociación TLS selecciona una de las cipher suites ofrecidas por el cliente, y el resultado puede verse en la sección `SSL-Session` de la salida por pantalla. El Alg. 57 ilustra este el caso de configurar tres cipher suites desde el lado del cliente. Se incluye también un extracto de la salida. Debido a que no se especificaron cipher suites post-cuánticas, dicha negociación no incluye algoritmos NTRU ni QSH.

```
# Ejecución del cliente:
./examples/client/client -l ECDHE-RSA-AES256-SHA:ECDHE-
   ECDSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA

# Fragmento de la salida:
[...]
SSL-Session:
  Protocol  : TLSv1.2
  Cipher    : TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
[...]
```

Algoritmo 57: wolfSSL. Forzado de cipher suites.

IX.2.3 Negociación QSH

wolfSSL implementa la cipher suite QSH (ver Sección III.2) para proveer algoritmos post-cuánticos en la negociación TLS. Con la intención de entender la implementación de wolfSSL en Cliente y Servidor, y para facilitar la identificación del uso de QSH durante el intercambio TLS, se modificó el servidor agregando la línea $469 - 470$ mostrada en el Alg. 58. La Fig. 30 muestra la nueva salida del servidor luego de establecer la conexión.

```
466 /* Archivo: examples/server/server.c */
467 if (ret > 0) {
468   input[ret] = 0; /* null terminate message */
469   printf("Quantum-Safe Handshake Ciphersuite?? -> %s\n",
470     (wolfSSL_isQSH(ssl))?"Yes":"No");
471   printf("Client message: %s\n", input);
472 }
```

Algoritmo 58: wolfSSL. Comprobación QSH Servidor.

```
root@juncotic-lubuntu:/opt/wolfssl/wolfssl-git# ./examples/server/server
peer's cert info:
 issuer : /C=US/ST=Montana/L=Bozeman/O=wolfSSL_2048/OU=Programming-2048/CN=www.wolfs
sl.com/emailAddress=info@wolfssl.com
 subject: /C=US/ST=Montana/L=Bozeman/O=wolfSSL_2048/OU=Programming-2048/CN=www.wolfs
sl.com/emailAddress=info@wolfssl.com
 serial number:aa:c4:bf:4c:50:bd:55:77
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
SSL curve name is FFDHE_3072
Server Random : AC90223F051262CD53DBE916D9B131C6D877675A39AF2D98D6FF099F967DF54B
Quantum-Safe Handshake Ciphersuite?? -> No
Client message: hello wolfssl!
```

Figura 30: Evaluando QSH en SSL/TLS. Caso negativo.

Ahora bien, si el cliente ofrece la cipher suite `QSH`, el servidor la utiliza de manera adicional a una cipher suite tradicional. En el Alg. 59 se ve el comando ejecutado por el cliente, donde se agrega la cipher suite `QSH`. Ahora el registro de conexión muestra el mensaje afirmativo para QSH (Fig. 31).

```
    cd /opt/wolfssl/wolfssl-git
    ./examples/client/client -l ECDHE-RSA-AES256-SHA:ECDHE-
       ECDSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA:QSH
```

Algoritmo 59: wolfSSL. Forzado de QSH en el cliente.

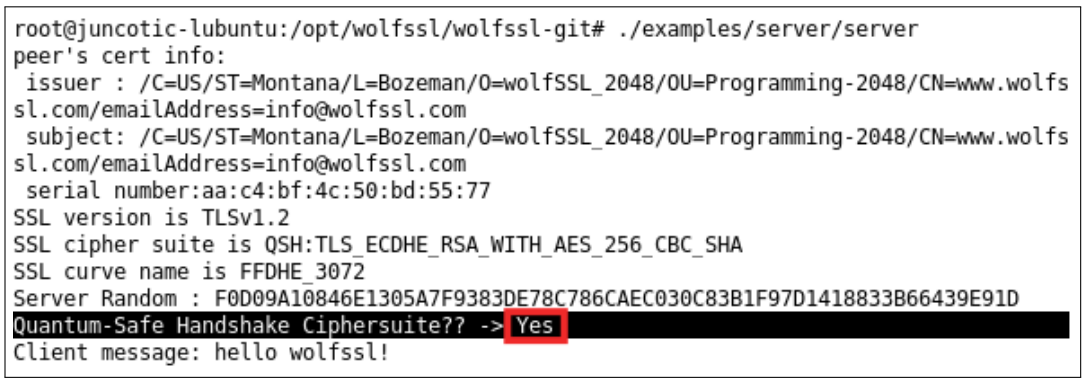

Figura 31: Evaluando QSH en SSL/TLS. Caso positivo.

Otras opciones interesantes y útiles que pueden utilizarse en el servidor para realizar pruebas son las siguientes:

- **-x**: Permite imprimir los mensajes de error sin cerrar la conexión.
- **-b**: Atiende el servicio en todas las interfaces de red, no sólo localhost.
- **-i**: Permite mantener el servidor activo para realizar múltiples conexiones desde los clientes.
- **-n**: Habilita el cifrado NTRU en SSL/TLS (ver más adelante).
- **-c**: Especifica la ruta al certificado digital a utilizar en la conexión.
- **-k**: Especifica la ruta a la clave privada a utilizar en la conexión.

El cliente también acepta otras opciones que se documentan a continuación, aunque serán de utilidad más adelante:

- **-l**: Ya mencionada, permite especificar una cipher suite determinada.
- **-h**: Permite especificar, por ip o nombre, el host a conectar.
- **-p**: Permite especificar un puerto destino distinto de 11111.
- **-d**: Desactiva la verificación de par, lo que facilita las pruebas con certificados digitales autofirmados.
- **-g**: Envía un mensaje `HTTP GET` al servidor, útil para https.

IX.2.4 Negociación NTRU

La suite wolfSSL provee, junto con sus archivos de código fuente y configuraciones, una serie de claves y certificados digitales almacenados en el directorio `certs/` de la raíz del repositorio. Dentro del mismo se encuentran una clave y un certificado digital generados utilizando el algoritmo NTRU:

```
    certs/ntru-cert.pem    # Certificado digital
    certs/ntru-key.raw     # Clave privada
```

El binario del servidor permite activar el algoritmo NTRU añadiendo la opción `-n`. Las opciones `-c` y `-k` se utilizan para especificar certificado y clave respectivamente, como se ve en el Alg. 60.

```
    cd /opt/wolfssl/wolfssl-git
    ./examples/server/server -c ./certs/ntru-cert.pem \
      -k ./certs/ntru-key.raw -n
```

Algoritmo 60: wolfSSL. Servidro NTRU.

Luego se lanzó el cliente sin especificar ningún parámetro adicional, y de manera predeterminada estableción la conexión TLS utilizando NTRU, como puede verse en la Fig. 32. El registro de la conexión en el servidor se ve en la Fig. 33.

```
root@juncotic-lubuntu:/opt/wolfssl/wolfssl-git# ./examples/client/client
Session Ticket CB: ticketSz = 142, ctx = initial session
peer's cert info:
 issuer : /C=US/ST=Montana/L=Bozeman/O=Sawtooth/OU=Consulting/CN=www.wolfssl.com/ema
ilAddress=info@wolfssl.com
 subject: /C=US/ST=Orgeon/L=Portland/SN=Test/O=wolfSSL/OU=Development/CN=www.wolfssl
.com/emailAddress=info@wolfssl.com
 serial number:94:68:8b:78:9e:1e:8b:e9:97:91:cf:80:21:bf:2d
SSL version is TLSv1.2
SSL cipher suite is TLS_NTRU_RSA_WITH_AES_256_CBC_SHA
Session timeout set to 300 seconds
Client Random : 5E2EE5200C69074549975D7F233200BB09A996208A3633706AF6A06CAFE635FC
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : TLS_NTRU_RSA_WITH_AES_256_CBC_SHA
    Session-ID:
    Session-ID-ctx:
```

Figura 32: Negociación NTRU en TLS v1.2 - Cliente.

```
root@juncotic-lubuntu:/opt/wolfssl/wolfssl-git# ./examples/server/server -c ./certs/
ntru-cert.pem -k ./certs/ntru-key.raw -n
peer's cert info:
 issuer : /C=US/ST=Montana/L=Bozeman/O=wolfSSL_2048/OU=Programming-2048/CN=www.wolfs
sl.com/emailAddress=info@wolfssl.com
 subject: /C=US/ST=Montana/L=Bozeman/O=wolfSSL_2048/OU=Programming-2048/CN=www.wolfs
sl.com/emailAddress=info@wolfssl.com
 serial number:aa:c4:bf:4c:50:bd:55:77
SSL version is TLSv1.2
SSL cipher suite is TLS_NTRU_RSA_WITH_AES_256_CBC_SHA
SSL curve name is FFDHE_3072
Server Random : E781A60ABAC53769F17FAAD1D75F6BDA596852816C4A7DB29E44A26F750AB2A7
Quantum-Safe Handshake Ciphersuite?? -> No
Client message: hello wolfssl!
root@juncotic-lubuntu:/opt/wolfssl/wolfssl-git#
```

Figura 33: Negociación NTRU en TLS v1.2 - Servidor.

Como puede verse en la salida del servidor (Fig. 33), al negociar una cipher-suite NTRU en SSL/TLS, no se activa QSH, un algoritmo que se habilitaba anteriormente de manera predeterminada (ver Figuras 28 y 29).

Surgió en este punto la duda de si wolfSSL, al habilitar NTRU, desactiva QSH, o desactiva cualquier otra cipher suite distinta de NTRU. Se intentó conectar un cliente especificando una cipher suite estándar no NTRU, y el servidor rechazó la negociación SSL/TLS al no poder verificar la cipher suite, como puede verse en la Fig. 34.

```
root@juncotic-lubuntu:/opt/wolfssl/wolfssl-git# ./examples/server/server -c ./certs/
ntru-cert.pem -k ./certs/ntru-key.raw -n -x

SSL_accept error -501, can't match cipher suite
wolfSSL error: SSL_accept failed
Continuing server execution...

peer has no cert!
SSL version is TLSv1.2
SSL cipher suite is NONE
SSL curve name is FFDHE_3072
wolfSSL error: SSL in error state
Continuing server execution...

Server Random : 0000000000000000000000000000000000000000000000000000000000000000
SSL_read input error -501, can't match cipher suite
wolfSSL error: SSL_read failed
Continuing server execution...
```

Figura 34: Servidor en modo NTRU recibiendo otras cipher suites.

Al invocarse al cliente sin utilizar ninguna cipher suite, se negocia NTRU con el servidor. Incluso si se fuerza la negociación de QSH junto con cipher suites NTRU, QSH no se habilita. Por ejemplo, si el cliente especifica la cipher suite `NTRU-AES256-SHA:QSH`, el cifrador resultante de la negociación SSL/TLS será `TLS_NTRU_RSA_WITH_AES_256_CBC_SHA`, que corresponde a la cipher suite propuesta por el cliente, omitiendo QSH, cipher suite que fue especificada explícitamente. El Alg. 61 representa un fragmento de la salida del servidor que ejemplifica el caso.

```
SSL version is TLSv1.2
SSL cipher suite is TLS_NTRU_RSA_WITH_AES_256_CBC_SHA
SSL curve name is FFDHE_3072
Server Random : 7148
   D9E984C5C4C8A6958B701C93608744194A93B37D408DE986...
Quantum-Safe Handshake Ciphersuite?? -> No
```

Algoritmo 61: wolfSSL. Salida NTRU del servidor.

Este comportamiento es debido a que wolfSSL activa la cipher suite QSH como un complemento resistente a ataques cuánticos junto con algoritmos tradicionales. NTRU ya dispone de un intercambio post-cuántico integrado en la librería, y se activa automáticamente al establecer la conexión.

La cipher suite QSH tiene mayor preferencia que otros protocolos comunes durante la selección de algoritmos. Si un usuario integra NTRU dentro de wolfSSL y ambos extremos de conexión soportan NTRU, entonces la cipher suite NTRU y su intercambio de claves seguro serán seleccionados de manera predeterminada. Sólo en el caso de que el usuario excluya explícitamente a NTRU se activa QSH y se combina con cipher suites tradicionales. El manual de usuario de wolfSSL (WolfSSL, 2021) menciona que NTRU acelera el proceso de conexión SSL/TLS de 20 a 200 veces sobre un intercambio RSA, y la mejora se incrementa según aumenta el tamaño de la clave utilizada. Es decir, el proceso de negociación de SSL/TLS será más rápido con claves NTRU de 8192 bits que con claves de 1024 bits RSA.

Finalmente, debido a que NTRU no tiene una adopción masiva como ocurre con RSA, wolfSSL lo implementa en modo híbrido con otras cipher suites. Los desarrolladores de wolfSSL propusieron la inclusión de NTRU en modo híbrido dentro de QSH, y así lograr que TLS utilice pares de claves post-cuánticas NTRU por única vez, además de los pares de claves de algoritmos tradicionales (Whyte et al., 2017b; WolfSSL, 2019).

IX.3 WOLFSSL Y SERVICIOS HTTPS: APACHE

Esta sección detalla los experimentos realizados integrando wolfSSL con Apache para montar un servidor HTTPS post-cuántico. Para llevar a cabo esta prueba de concepto fue necesario el código fuente de wolfSSL compilado convenientemente para permitir su integración, y una versión relativamente actualizada de Apache. Por recomendación de los desarrolladores de wolfSSL que colaboraron con su soporte, se utilizó la versión 2.4.42-dev (Unix) de Apache, descargada directamente desde el repositorio SVN del proyecto. Cabe aclarar que wolfSSL tiene soporte para su integración con versiones de Apache posteriores a la v2.4.0(WolfSSL, 2020b).

Primeramente fue necesario instalar las librerías de parseo de archivos xml, necesarias para la compilación de Apache. Luego se descargó y compiló Apache desde su repositorio. Esto se realizó en el directorio `/opt/wolfssl` creado a tal efecto. Los pasos pueden observarse en el Alg. 62.

```
# Instalación de dependencias:
sudo apt install libxml2 libxml2-dev

# Descarga de Apache httpd:
cd /opt/wolfssl/wolfssl-git/
svn checkout https://svn.apache.org/repos/asf/httpd/httpd/
   branches/2.4.x httpd-2.4.x/
cd httpd-2.4.x/
```

Algoritmo 62: wolfSSL y Apache. Dependencias y descarga.

Dado que el motor SSL/TLS de Apache está desarrollado para su integración con OpenSSL, se debió modificar y adaptar su código para integrarlo con wolfSSL. Estas modificaciones se realizaron sobre el archivo de código fuente `modules/ssl/ssl_engine_init.c`. Los cambios incluyeron modificaciones en los nombres de las funciones `SSL_CTX_*` por su equivalente en wolfSSL, `wolfSSL_CTX*`. En el repositorio que acompaña a la presente tesis se encuentra el archivo `diff` y el `ssl_engine_init.c` resultante.

wolfSSL implementa gran cantidad de funcionalidades SSL, de las cuales solo algunas fue necesario modificar en Apache para lograr la integración en la prueba de concepto básica. Para facilitar la identificación de estas funciones y su secuencia se utilizó como base el código del ejemplo `server.c`. Se realizó la compilación utilizando el modificador `-E` en `gcc` como se muestra en el Alg. 63, lo que permitió interpretar las directivas de preprocesamiento, y obtener el archivo `server.i`, una versión final del código que implementa NTRU. Este archivo intermedio se encuentra disponible en el repositorio Git que acompaña al presente trabajo (Córdoba, 2022).

```
cd /opt/wolfssl/wolfssl-git
gcc examples/server/server.c -o /tmp/server.i -I. -E
```

Algoritmo 63: wolfSSL. Preprocesado del servidor de ejemplo.

Además se agregaron dos líneas de depuración para verificar la lectura de la clave NTRU del servidor Apache. Este cambio puede verse en el Alg. 64.

```
1296 /* Archivo: /etc/wolfssl/wolfssl-git/httpd-2.4.x/modules/
        ssl/ssl_engine_init.c */
1297 if (wolfSSL_CTX_use_NTRUPrivateKey_file(mctx->ssl_ctx,
        keyfile) == WOLFSSL_SUCCESS)
1298   printf("DEBUG wolfSSL_CTX_use_NTRUPrivateKey_file
        SUCCESS!!!\n");
```

Algoritmo 64: wolfSSL y Apache. Salida NTRU del servidor.

Luego de realizados estas modificaciones se procedió a generar los scripts de soporte para la compilación y posterior instalación de Apache. Esto se llevó a cabo mediante los comandos mostrados en el Alg. 65

```
cd /etc/wolfssl/wolfssl-git/httpd-2.4.x
./buildconf
./configure \
  --enable-access_compat=shared --enable-actions=shared \
  --enable-alias=shared --enable-allowmethods=shared \
  --enable-auth_basic=shared --enable-authn_core=shared \
  --enable-authn_file=shared --enable-authz_core=shared \
  --enable-authz_groupfile=shared --enable-mime=shared \
  --enable-authz_host=shared --enable-authz_user=shared \
  --enable-autoindex=shared --enable-dir=shared \
  --enable-env=shared --enable-headers=shared \
  --enable-include=shared --enable-log_config=shared \
  --enable-negotiation=shared --enable-proxy=shared \
  --enable-proxy_http=shared --enable-rewrite=shared \
  --enable-setenvif=shared --enable-unixd=shared \
  --enable-ssl=shared --enable-ssl-staticlib-deps \
  --enable-mods-static=all --enable-static-ab \
  --with-included-apr --with-libxml2 \
  --with-wolfssl=/opt/wolfssl/wolfssl-git --enable-ssl
make && make install
```

Algoritmo 65: wolfSSL y Apache. Compilación e instalación.

La última línea de configuración invoca la opción `--with-wolfssl`, que indica el directorio de los fuentes de wolfSSL que se utilizará para compilar Apache. Debe recordarse que esta integración es bidireccional, durante la compilación de wolfSSL también se especificó el soporte para su integración con https, tal y como se mencionó en la Sección IX.2.

Los últimos dos comandos, `make` y `make install`, realizaron la compilación e instalación respectivamente. La ejecución fue satisfactoria y esta implementación de servidor HTTPS se instaló en el directorio predeterminado `/usr/local/apache2`. En la Fig. 35 puede verse la salida del comando `ldd` que muestra al binario de Apache, `httpd`, correctamente enlazado con las librerías de wolfSSL y NTRU.

Luego se editó el archivo de configuración de Apache, `conf/httpd.conf`, para habilitar el módulo SSL. Para ello se activó la línea que incluye la confi-

```
root@juncotic-lubuntu:/usr/local/apache2# ldd bin/httpd
        linux-vdso.so.1 (0x00007fffc57d0000)
        libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007fe8581d6000)
        libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007fe8581ba000)
        libapr-2.so.0 => /usr/local/apache2/lib/libapr-2.so.0 (0x00007fe85814d000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fe85812c000)
        libwolfssl.so.24 => /usr/local/lib/libwolfssl.so.24 (0x00007fe857f45000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe857d5a000)
        libuuid.so.1 => /lib/x86_64-linux-gnu/libuuid.so.1 (0x00007fe857d4f000)
        libcrypt.so.1 => /lib/x86_64-linux-gnu/libcrypt.so.1 (0x00007fe857d15000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fe857d0f000)
        libxml2.so.2 => /lib/x86_64-linux-gnu/libxml2.so.2 (0x00007fe857b67000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fe8583f3000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fe857a19000)
        libntruencrypt.so.0 => /usr/local/lib/libntruencrypt.so.0 (0x00007fe857a0a000)
        libicuuc.so.63 => /lib/x86_64-linux-gnu/libicuuc.so.63 (0x00007fe857839000)
        liblzma.so.5 => /lib/x86_64-linux-gnu/liblzma.so.5 (0x00007fe857812000)
        libicudata.so.63 => /lib/x86_64-linux-gnu/libicudata.so.63 (0x00007fe855e22000)
        libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007fe855c40000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fe855c25000)
```

Figura 35: Librerías compiladas en httpd.

guración del módulo `httpd-ssl`. Además se activó la entrada de `ServerName` para evitar advertencias en tiempo de ejecución como se ve en el Alg. 66.

```
    # (Dentro de /etc/wolfssl/wolfssl-git/httpd-2.4.x)

    # Archivo conf/httpd.conf:
    ServerName www.juncotic.com:80
    Include conf/extra/httpd-ssl.conf

    # Archivo conf/extra/httpd-ssl.conf:
    # SSLHonorCipherOrder on
    SSLCertificateFile "/opt/keys/wolfssl/ntru-cert.pem"
    SSLCertificateKeyFile "/opt/keys/wolfssl/ntru-key.raw"
```

Algoritmo 66: wolfSSL y Apache. Configuraciones del servidor web.

El mismo Alg. 66 muestra la configuración de SSL/TLS de Apache, dentro del archivo `conf/extra/httpd-ssl.conf`. Allí se editaron las rutas al certificado y clave privada provistos como ejemplo por wolfSSL, y se desactivó la opción `SSLHonorCipherOrder` por no estar soportada aún por el motor SSL/TLS de wolfSSL.

Se copiaron los certificados y claves de wolfSSL, originalmente situados en el directorio `wolfssl/certs`, dentro de `/opt/keys/wolfssl`, donde se almacenan todas las claves utilizadas en el presente proyecto.

Se inició el servidor Apache invocando directamente al binario `bin/httpd`. Se utilizó la opción `-X` para ejecutarlo en modo de depuración, y de esta forma visualizar los mensajes de error en el caso de producirse fallos. Como dato adicional se menciona que dicho binario permite aumentar la cantidad de mensajes de depuración agregando la opción `-e DEBUG`. De manera predeterminada, el servidor quedó atendiendo en el puerto TCP 443. Se utilizó el cliente de ejemplo de wolfSSL utilizado en la sección anterior, esta vez especificando el puerto donde atiende el servidor Apache. Las líneas de ejecución de ambos pueden verse en el Alg. 67.

```
# Ejecución del servidor Apache:
cd /usr/local/apache2
./bin/httpd -X

# Ejecución del cliente wolfSSL:
./examples/client/client -p 443 -g -d
```

Algoritmo 67: wolfSSL y Apache. Ejecución de servidor y cliente https.

En la Fig. 36 puede verse el servidor Apache corriendo en la terminal superior, y el cliente conectado en la inferior. En la terminal del cliente se aprecia la cipher suite NTRU negociada entre ambos para TLSv1.2.

Si bien se ha logrado la conexión cliente-servidor utilizando el cliente provisto por wolfSSL y el servidor Apache, en algunos casos aleatorios se detectaron fallas en la negociación, y terminaciones abruptas del proceso servidor. El Alg. 68 muestra la respuesta del cliente y servidor en el momento de producirse uno de estos fallos.

```
# Ejecución y salida del cliente (fallo):
cd /etc/wolfssl/wolfssl-git/
./examples/client/client -p 443 -g -d
CRL callback url =
wolfSSL_connect error -308, error state on socket
wolfSSL error: wolfSSL_connect failed

# Salida del servidor (segfault)
Segmentation fault (core dumped)
```

Algoritmo 68: wolfSSL y Apache. Errores detectados.

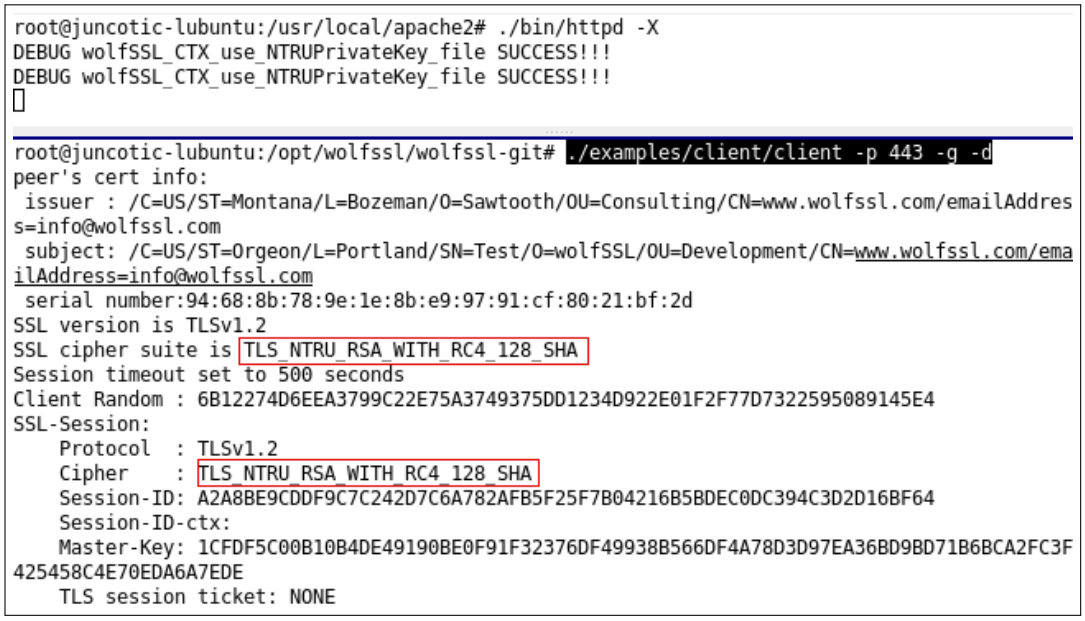

Figura 36: Conexión NTRU en Apache + wolfSSL.

El servidor registra los eventos de conexión con gran nivel de detalle en el archivo `log/error_log` local de la implementación. La información relativa a este fallo en el servidor arrojó el mensaje mostrado en el Alg. 69.

```
# Archivo: /etc/wolfssl/wolfssl-git/httpd-2.4.x/log/
    error_log

[Fri Jun 26 21:07:16.448582 2020] [core:notice] [pid
    32013:tid 139640733110464] AH00051: child pid 32484
    exit signal Segmentation fault (11), possible coredump
    in /usr/local/apache2

[Fri Jun 26 21:09:44.250532 2020] [ssl:warn] [pid 1512:tid
     139988311048384] AH01882: Init: this version of
    mod_ssl was compiled against a newer library (4.3.0,
    version currently loaded is 4.3.0) - may result in
    undefined or erroneous behavior
```

Algoritmo 69: wolfSSL y Apache. Info de errores detectados.

Este error se produjo por conflictos en la integración de wolfSSL con el módulo `mod_ssl` de Apache. Esta es una de las causas por las que esta integración continúa siendo experimental al momento de realizar estas pruebas, y no se recomienda su utilización en servicios en producción.

IX.4 WOLFSSL Y SERVICIOS OPENVPN

Esta sección describe la integración de wolfSSL con OpenVPN. Si bien wolfSSL provee soporte para compatibilidad con OpenVPN, ésta integración resulta precaria y aún no brinda, al día de la fecha, la posibilidad de utilizar protocolos de encapsulamiento ni cipher suites post-cuánticos. Igualmente a continuación se muestra la prueba de concepto realizada utilizando algoritmos tradicionales en TLS v1.3 con la intención de profundizar en los detalles y limitaciones de dicha integración.

Compilación e integración de wolfSSL y OpenVPN

Se re-compiló wolfSSL para añadir compatibilidad con OpenVPN. Por esta razón se creó un directorio específico para integrar estas aplicaciones, y se replicó luego dicho directorio en el equipo cliente para realizar la conexión. El Alg. 70 muestra los pasos llevados a cabo para compilar e instalar wolfSSL. Según la documentación oficial (Sosinowicz, 2020) la compatibilidad con OpenVPN se logra utilizando la opción `--enable-openvpn`, que se añadió a las ya empleadas en la Sección IX.

```
# Creación del directorio base
mkdir /opt/wolfssl_openvpn/
cd /opt/wolfssl_openvpn/

# Clonado del repositorio Git de wolfSSL
git clone https://github.com/wolfSSL/wolfssl.git
cd wolfssl

# Configuración, compilación e instalación de wolfSSL
./autogen.sh
./configure --enable-qsh --with-ntru=/usr/local/lib \
-enable-opensslextra --enable-supportedcurves \
--enable-apachehttpd --enable-tlsx --enable-all \
--enable-ecc --enable-psk --enable-aesccm \
C_EXTRA_FLAGS="-DWOLFSSL_STATIC_RSA -DHAVE_SNI" \
--enable-openvpn
make
sudo make install
```

Algoritmo 70: wolfSSL y OpenVPN. Compilación de wolfSSL.

Para compilar OpenVPN se utilizó un fork adaptado para su integración con wolfSSL. Dentro del mismo directorio creado en el paso anterior se clonó el repositorio de OpenVPN, y se cargó la rama `wolfssl-compat` mediante el comando `checkout` de `git`. Luego se generaron los archivos de configuración y se compiló la implementación de OpenVPN. Debe notarse que el script `configure` recibe como argumento una nueva opción para indicar que la compilación debe realizarse utilizando wolfSSL en lugar de la suite OpenSSL. El Alg. 71 detalla estos pasos.

En este caso particular no se instaló OpenVPN en el sistema operativo para evitar generar conflictos con otras implementaciones y versiones. Igualmente el binario de OpenVPN generado pudo invocarse directamente desde su directorio de fuentes, y mostrado en el mismo Alg. 71.

```
# Clonado del repositorio Git de OpenVPN
cd /opt/wolfssl_openvpn/
git clone https://github.com/julek-wolfssl/openvpn.git
cd openvpn
git checkout wolfssl-compat

# Configuración y compilación de OpenVPN
autoreconf -i -v -f
./configure --with-crypto-library=wolfssl
make

# Directorio de fuentes de OpenVPN:
cd /opt/wolfssl_openvpn/openvpn/src/
```

Algoritmo 71: wolfSSL y OpenVPN. Compilación de OpenVPN.

OpenVPN/wolfSSL - Pruebas de concepto

Partiendo de la implementación de OpenVPN compilada en la sección anterior, se procedió a realizar la prueba de concepto. En primer lugar puede verificarse que dicho OpenVPN está integrado con la librería TLS de WolfSSL utilizando el modificador `--version`, como puede observarse en la Fig. 37.

Los algoritmos de cifrado soportados son variantes de AES con claves de 128, 192 y 256 bits, y de DES y 3DES, todos en modo CBC. Como se men-

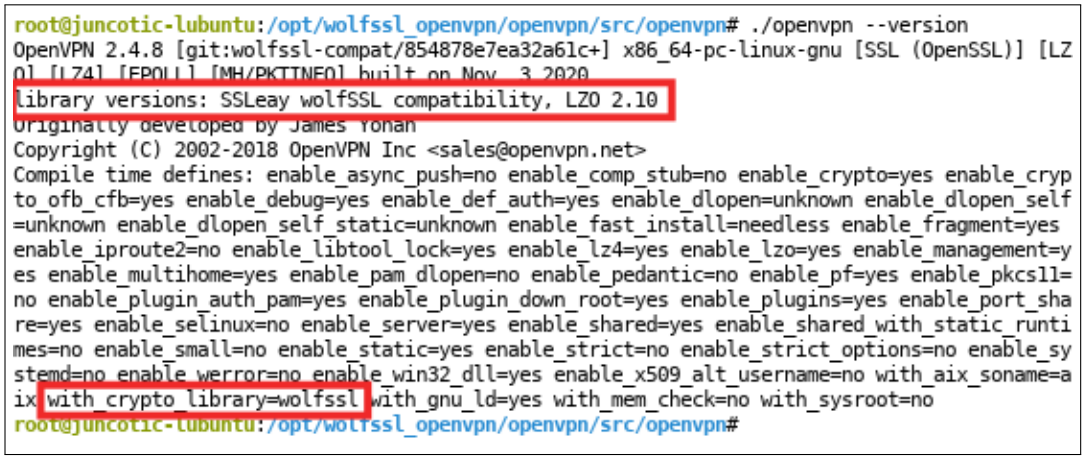

Figura 37: OpenVPN/wolfSSL. Información de versión.

cionó anteriormente, no soporta cipher suites de TLS en ninguna de sus versiones. Esta información puede extraerse utilizando los modificadores `--show-ciphers` y `--show-tls` respectivamente (véase Fig. 38).

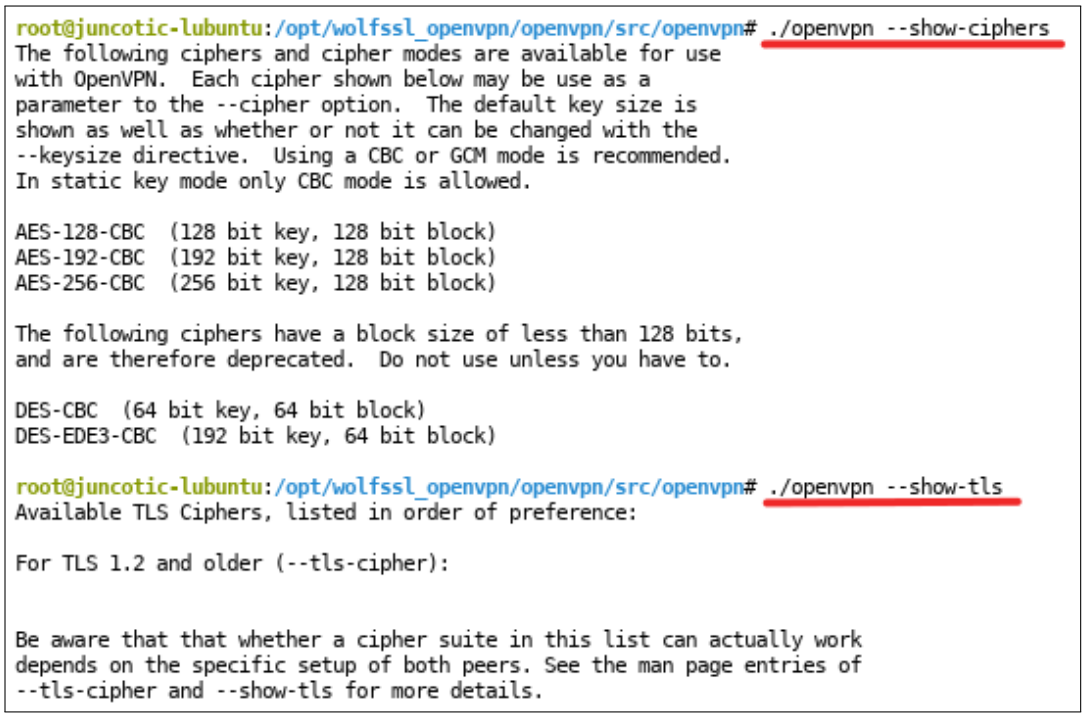

Figura 38: OpenVPN/wolfSSL. Cipher suites y algoritmos TLS.

Puede anticiparse que esta versión de OpenVPN, al momento de realizar el presente trabajo, resulta sumamente reducida en comparación con su par PQCrypto-VPN analizado en la Sección VIII.7.

Las pruebas de conexión se realizaron referenciando los certificados y claves RSA utilizados en la Sección VIII.7 para PQCrypto-VPN. Los resultados fueron satisfactorios, se omiten en el presente trabajo por no considerarse relevantes para el objetivo del mismo.

Este experimento dio cuenta de que la integración wolfSSL y OpenVPN es posible y la compatibilidad entre ambas implementaciones no genera errores ni inconsistencias para experimentos simples. No obstante, los algoritmos de cifrado de tráfico utilizados son escasos comparados con su contraparte OpenSSL, y no soporta ninguna cipher suite TLS.

A diferencia de la adaptación de Apache para su uso con wolfSSL realizada en la Sección IX.3, en este caso OpenVPN ya es compatible con wolfSSL, y la adaptación al uso de criptografía NTRU (u otros algoritmos post-cuánticos) requiere, por un lado, de la incorporación de cipher suites TLS en la integración, y por otro, de la adaptación de las mismas mediante el agregado de algoritmos post-cuánticos como NTRU.

Si bien OpenVPN y wolfSSL por separado son implementaciones ampliamente utilizadas y probadas, su integración se encuentra en etapas muy tempranas de desarrollo, por lo que su realización excede los alcances del presente trabajo de tesis.

X OTRAS IMPLEMENTACIONES

Durante el transcurso de la investigación llevada a cabo para completar el presente trabajo se produjeron importantes cambios tanto en las bibliotecas de código que proveen criptografía post-cuántica como en las implementaciones de software que las utilizan. Así, hubieron proyectos que al inicio de esta investigación se creían prometedores, que luego dejaron de actualizarse, y otros que en su momento no existían y hoy representan interesantes propuestas.

Entre los proyectos que quedaron obsoletos se cuenta OpenSSL-RingLWE (Singh, 2015), reseñado más adelante en este capítulo, TinySSH (Mojžíš, 2018) como alternativa post-cuántica a OpenSSH (OpenSSH, 2021), o Codecrypt (Kratochvil, 2018) como reemplazo de GNUPG (The GnuPG Project, 2021). Los siguientes apartados mencionarán algunas características de las implementaciones más relevantes.

X.1 OQS-BORINGSSL

Google originalmente utilizó OpenSSL para proveer de seguridad SSL/TLS a sus plataformas de software. Durante años desarrolló y mantuvo parches para OpenSSL, y los integraba en sus productos. Con el incremento de la cantidad y complejidad de los productos y servicios de Google, el esfuerzo para realizar y mantener estos parches fue cada vez mayor. Así fue que decidieron crear un fork de OpenSSL que se adaptara mejor a sus líneas de desarrollo. Este fork se denominó BoringSSL (Google Inc., 2020). Se trata de un proyecto de código abierto optimizado por Google para sus plataformas de software, no es una implementación de SSL/TLS de propósito

general como OpenSSL, y no se recomienda su uso por terceros debido a que que su API no es estable y recibe cambios frecuentes.

El proyecto OQS realizó un fork de esta aplicación y le integró liboqs para proveerle algoritmos de autenticación y de intercambio de claves post-cuánticos, dando origen a OQS-BoringSSL (OQS, 2020). Como otras implementaciones de OQS, se trata de un proyecto experimental para prototipado de soluciones resistentes a ataques cuánticos, y no es recomendable para servicios en producción.

Si bien los algoritmos soportados por esta implementación no están aún estandarizados por el NIST, no existen vulnerabilidades reportadas para los mismos. No obstante, al igual que con OQS-OpenSSL, la principal recomendación es utilizar algoritmos híbridos para lograr, como mínimo, una seguridad equivalente a la del cifrado tradicional.

X.2 OPENSSL - RINGLWE

Una implementación que en su momento fue considerada para analizar detalladamente en el presente trabajo es OpenSSL-RingLWE (Singh, 2015). El estudio finalmente fue descartado debido a que el proyecto tuvo su última actualización en mayo de 2016.

OpenSSL-RingLWE es un fork de OpenSSL v1.0.2e, y por ello hoy se considera obsoleto. Provee seguridad post-cuántica utilizando Ring-LWE, una eficiente alternativa a Diffie-Hellman para intercambio de claves que se supone segura contra ataques de computadoras cuánticas y tradicionales. Se basa en problemas de anillos de aprendizaje con errores (RLWE, por sus siglas en inglés) y define dos cipher suites:

- RLWE-RSA-AES128-GCM-SHA256
- RLWEP-RSA-AES128-GCM-SHA256

Ambos brindan seguridad híbrida de 256 bits con RSA, el primero de ellos lo hace utilizando claves públicas de 16384 bits, mientras que el segundo usa claves de 13120 bits. El proyecto quedó suspendido antes de poder implementar las cipher suites basadas en ECDSA. La implementación además era altamente parametrizable usando la biblioteca libcrypt con diferentes niveles de seguridad para RLWE. Si bien el proyecto se encuentra discontinuado, representa una de las pocas alternativas basadas en OpenSSL que no dependen de las librerías liboqs.

X.3 VPN WIREGUARD

Wireguard (Donenfeld, 2017) es una aplicación y protocolo de comunicación que permite establecer redes privadas virtuales, liberada bajo licencia de software libre GPLv2. Corre como un módulo del núcleo Linux o BSD, y tiene como objetivo ofrecer mejor rendimiento, mayor tolerancia a fallos y configuración más amigable respecto de aplicaciones ampliamente utilizadas como OpenVPN e IPSec. La primer versión estable para Linux se liberó en Marzo de 2020, y su desarrollo se encuentra activo, por lo que se espera crezca a corto plazo y se extienda su uso.

Wireguard implementa el intercambio o handshake Noise_IK (Perrin, 2018), y todo el tráfico de red que genera viaja sobre transporte UDP. Utiliza el protocolo ChaCha20 para cifrado simétrico, ECDH con Curve25519 para el intercambio de claves inicial, y hashes basados en BLAKE2. Su intercambio de claves facilita el control de suplantación de identidad, evita los ataques de reenvío (replay attacks) y brinda PFS, entre otras.

El protocolo Wireguard no cuenta con criptografía post-cuántica nativa, ni integra librerías externas como liboqs. Sin embargo, sí añade con una capa de criptografía simétrica opcional para utilizar en combinación con los

algoritmos de clave pública, y que puede integrar fácilmente mecanismos post-cuánticos al intercambio de claves (Donenfeld, 2020).

X.4 IPSEC

IPSec (Frankel y Krishnan, 2011) es uno de los conjuntos de protocolos de redes privadas virtuales en capa de red más importante y confiable en la actualidad. Entre los protocolos que utiliza IPSec se encuentra IKEv2 (Internet Key Exchange, Intercambio de claves en Internet) (Kaufman, 2005) para la negociación inicial del canal seguro. IKE realiza un intercambio de claves basado en ECDH, un algoritmo que, como se mencionó, se considera vulnerable al criptoanálisis cuántico.

En 2017 se presentó un borrador de estándar a la IETF (Fluhrer et al., 2017) que luego fue reemplazado por la RFC8784 en junio de 2020 (Fluhrer et al., 2020). El documento plantea la incorporación de una clave pre-compartida post-cuántica (PPK, por sus siglas en inglés) junto con la negociación de DH o ECDH. Así, ambos extremos de la comunicación deberán soportar PPK para establecer el túnel. En el caso de que uno de los extremos no lo soporte, se establecería un túnel IPSec tradicional.

Por su parte, y de forma independiente, StrongSwan, una de las implementaciones de IPSec de código abierto más activas, propuso en 2015 la incorporación de NTRU en la negociación de IKE a través de un plugin, de modo que podría lograr seguridad post-cuántica durante la primera etapa de establecimiento del túnel seguro (strongSwan, 2015).

X.5 ALTERNATIVAS SEGURAS A SSH

En esta sección se mencionarán algunas alternativas a OpenSSH que, de una u otra forma, permiten el agregado de algoritmos post-cuánticos en la comunicación segura.

XS: eXperimental Shell

XS (RLabs, 2020) es una alternativa simple a SSH escrita en lenguaje Go, de código abierto (licencia MIT) y cuyo desarrollo es activo. Permite sesiones remotas interactivas y no interactivas, transferencia de archivos, y ofuscación opcional del tráfico de red. Como algoritmos de intercambio de claves soporta tres variantes:

- Intercambio basado en HerraduraKEx (Herrera, 2019).
- Encapsulamiento de claves KYBER IND-CCA-2 (Ducas et al., 2019).
- Algoritmo de cifrado NewHope (Alkim et al., 2017).

KYBER es un mecanismo de encapsulamiento de claves (KEM) cuya seguridad se basa en la dificultad para resolver problemas de aprendizaje con errores (LWE, por sus siglas en inglés). Es un algoritmo post-cuántico basado en retículos. NewHope es un método de intercambio de claves también basado en Ring-LWE aceptado en la segunda ronda de estandarización del NIST, aunque no fue incluido en la tercera convocatoria.

OQS-OpenSSH

OQS-OpenSSH (OQS, 2018) es un fork de OpenSSH que incorpora criptografía post-cuántica integrando liboqs. Su última versión disponible es la v7.9. Esta implementación es experimental, y no es recomendable su uso en servicios de producción. Tiene las mismas limitaciones que se comentaron para otras integraciones de liboqs tales como OQS-OpenSSL, por lo que, al igual que en ellas, es recomendable el uso de criptografía híbrida en la negociación del túnel cifrado.

X.6 CRIPTOGRAFÍA POST-CUÁNTICA EN TOR

En (Tujner, 2019) se presenta un resumen de los mecanismos de cifrado e intercambio de claves resistentes a ataques cuánticos, explica la ar-

quitectura de la red TOR (The Onion Routing - Enrutamiento de Cebolla) (The TOR Project, 2021), y propone la incorporación de intercambio de claves post-cuántico en la conexión. Específicamente plantea el uso de mecanismos de encapsulamiento de claves asimétricas integrando liboqs en el software TOR. Realiza una prueba de concepto utilizando SweetOnions (LeonHeTheFirst, 2016) como implementación de TOR. La implementación de SweetOnions utilizada está desarrollada en Python 2.7, mientras que liboqs está escrita en C, y posee un wrapper en Python 3, por lo que debió utilizar el paquete *future* de Python para realizar la conversión de versiones, y de esta manera poder llevar a cabo la integración. Cabe mencionar que hoy se encuentra disponible un fork de SweetOnions reescrito en Python 3 (MaitreRenard, 2020), no presente al momento de la publicación de (Tujner, 2019).

TOR logra su seguridad utilizando una combinación de criptografía simétrica AES y mecanismos asimétricos RSA. Usa RSA para cifrar la clave secreta AES que se utiliza para dar seguridad al tráfico de red dentro del túnel TOR. (Tujner, 2019) plantea dos soluciones prácticas, una post-cuántica pura y otra híbrida. La solución post-cuántica hace uso de algoritmos resistentes para remplazar a RSA en la negociación del canal, mientras que la solución híbrida cifra la clave AES con un algoritmo post-cuántico previo al cifrado RSA, lo que le brinda, como mínimo, seguridad equivalente a la de RSA. Como algoritmos de encapsulamiento incorpora Frodo-640-AES, Frodo-640-SHAKE, Kyber512, NewHope-512-CCA, NTRU-HPS-2048-509 y Sike-p503.

X.7 CRIPTOGRAFÍA POST-CUÁNTICA EN CURL

Si bien ya se ha mencionado en la Sección VIII.5.1 una integración entre el cliente cURL y la biblioteca liboqs del proyecto OQS como parte de una

serie de imágenes Docker provistas por OQS (Stebila et al., 2022), cabe mencionar también un trabajo del equipo de desarrollo de wolfSSL que tiene la intención de integrar su implementación de TLS con cURL y la incorporación de algoritmos KEM resistentes a ataques cuánticos (WolfSSL, Inc., 2022a).

X.8 INTEGRACIÓN WOLFSSL + LIBOQS

Como se mencionó en la Sección IX, desde la versión v5.0.0 de wolfSSL, liberada en Noviembre de 2021, el proyecto discontinuó su integración tanto con NTRU como con QSH (véase el ChangeLog de (WolfSSL, 2020a)).

Desde entonces el equipo de wolfSSL ha continuado trabajando en soporte de algoritmos post-cuánticos en TLS 1.3, pero ahora utilizando como base liboqs, la biblioteca post-cuántica del proyecto OQS, haciendo uso de los algoritmos finalistas de la 3ra ronda de estandarización del NIST.

X.9 CRIPTOGRAFÍA NEURONAL

La criptografía neuronal (Kinzel y Kanter, 2002) es una rama de la criptografía que se basa en el estudio de algoritmos estocásticos basados en redes neuronales artificiales para su uso en el cifrado y el criptoanálisis.

Las redes neuronales artificiales se identifican por su capacidad para explorar soluciones de manera selectiva para un problema determinado. Por esto es que tiene un amplio uso en el campo del criptoanálisis. Una clase particular de red neuronal utilizada en este ámbito es la Máquina de Paridad de Árbol (TPM, por sus siglas en inglés), un tipo especial de red neuronal multicapa de propagación hacia adelante (feed-forward) (Michie et al., 1994; Kinzel y Kanter, 2002).

Los conceptos de aprendizaje mutuo, autoaprendizaje y comportamiento estocástico de las redes neuronales artificiales pueden usarse para diver-

sos campos de la criptografía, como el cifrado de clave pública, las funciones hash, la generación pseudo-aleatoria de números, o la distribución de claves mediante el uso de la sincronización mutua de redes neuronales.

Este último punto es de especial importancia en el presente trabajo. Como se ha estudiado hasta ahora, la mayor parte de los mecanismos de intercambio de claves están basados en el protocolo Diffie-Hellman. El intercambio basado en la sincronización de redes TPM no se fundamenta en complejidad computacional, por lo que podría considerarse, en primera instancia, un protocolo potencialmente seguro contra criptoanálisis cuántico (Javurek et al., 2013; Stypiski y Niemiec, 2021). Igualmente este protocolo tiene otras vulnerabilidades conocidas, explotables desde el criptoanálisis clásico.

CONCLUSIONES

En esta sección se exponen las conclusiones surgidas de los análisis y experimentaciones realizados, y se mencionan algunas posibles líneas de investigación que surgen del presente trabajo.

XI CONCLUSIONES

Los algoritmos criptográficos utilizados en la actualidad tienen ya décadas de estudio, pruebas y criptoanálisis, lo que brinda a los profesionales de la seguridad informática tranquilidad a la hora de utilizarlos para asegurar las comunicaciones en Internet. Los algoritmos post-cuánticos, por su parte, se encuentran recién transitando los primeros pasos en el camino de la estandarización, por lo que al día de hoy la gran mayoría de ellos no cuentan con una verificación exhaustiva que confirme una seguridad comparable a la de los algoritmos tradicionales. Además, las implementaciones de software que permiten llevar a la práctica estos mecanismos post-cuánticos hoy se encuentran mayormente en estado experimental, y muchas de ellas con un desarrollo muy activo.

El presente trabajo realizó un relevamiento de las implementaciones de software más prometedoras y cuyo objetivo es el de proveer a SSL/TLS de algoritmos resistentes al criptoanálisis cuántico. Un requisito para poder analizar las implementaciones de software y su comportamiento fue disponer de su código fuente. Por esta razón que se puso foco en aquellas implementaciones de software que han sido liberadas bajo licencias de software libre y/o código abierto. Se estudió por un lado la librería liboqs y su integración con OpenSSL, y por otro wolfSSL y sus avances en la incorporación de algoritmos post-cuánticos, puesto que se trata de una implementación orientada a dispositivos embebidos e IoT, un campo que presenta grandes perspectivas de crecimiento a corto y mediano plazo (Horwitz, 2021).

Se analizó la arquitectura y los algoritmos de encapsulamiento e intercambio de claves post-cuánticos soportados por liboqs. Se estudió OQS-OpenSSL, la integración de liboqs con OpenSSL en sus versiones 1.0.2 y 1.1.1.

OQS-OpenSSL v1.0.2 permitió analizar la negociación de cipher suites TLS v1.2, mientras que la v1.1.1 posibilitó el uso de algoritmos de encapsulamiento de claves en TLSv1.3. A su vez se analizó la integración de OpenSSL v1.1.1 con dos de las implementaciones de protocolo HTTP más utilizadas: Apache y Nginx. Finalmente se realizaron pruebas de la integración de ambas versiones de OQS-OpenSSL con OpenVPN utilizando PQCrypto-VPN.

Se analizó también la integración de wolfSSL con la biblioteca libntru, y se montó una prueba de concepto de TLSv1.2 haciendo uso de cipher suites post-cuánticas basadas en QSH y NTRU. Finalmente se integró wolfSSL con Apache para brindar un servicio HTTPS que incorpora algoritmos post-cuánticos. Para ello se modificó el código fuente de Apache adaptándolo a las funciones NTRU implementadas por wolfSSL. Además se estudió la integración entre wolfSSL y OpenVPN. En este punto se montó el servicio de VPN basado en wolfSSL nativo sin mecanismos post-cuánticos, ya que la incorporación de NTRU en OpenVPN se encuentra en etapas muy tempranas de desarrollo al momento de la realización de este trabajo.

Respecto a OQS-OpenSSL y su integración con Apache, Nginx y OpenVPN, los resultados demuestran estabilidad en las ejecuciones y pruebas. Si bien no es recomendable utilizar estos servicios en producción, se aprecia un estado de desarrollo maduro. No ocurre lo mismo con wolfSSL y su integración con NTRU y Apache. Se evidencian comportamientos erráticos e inestabilidad en las ejecuciones, lo que evidencia que, más allá de que el desarrollo de estas implementaciones sea activo, aún se encuentra en una etapa experimental y no se recomienda su uso en producción.

Si bien la mayoría de las pruebas realizadas con OQS-OpenSSL y wolfSSL pudieron llevarse a cabo, la integración de criptografía post-cuántica no resultó sencilla en algunos casos, y hubo necesidad de modificar y adaptar el código fuente de los servicios para lograr su funcionamiento de manera

satisfactoria. Por lo dicho anteriormente se concluye que, al momento de realizar este trabajo, las implementaciones analizadas no son aptas para montar soluciones post-cuánticas en servidores de producción.

Finalmente se mencionaron otras implementaciones que dan soporte a algoritmos de encapsulamiento de claves y cifrado post-cuántico, algunas de las cuales resultan prometedoras por su activo desarrollo.

Además de lo expuesto anteriormente los resultados obtenidos permiten apreciar una elevada actividad en el campo de estudio de la criptografía post-cuántica. Las conferencias anuales de criptografía PQCrypto (Bernstein y Lange, 2020) están dando un gran impulso al avance científico en este campo, y se convierten en el punto de encuentro de los investigadores.

Durante la realización de esta tesis se evidenció un alto grado de actividad en el desarrollo de las principales implementaciones de software analizadas, lo que da cuenta del interés, por parte de la comunidad de desarrolladores de software libre y de código abierto, por lograr avances prácticos en este campo.

Finalmente debe destacarse que la naturaleza FLOSS (Stallman, 2016) de las implementaciones de software utilizadas facilitó la realización de las pruebas de concepto. Esta ventaja permite además que gran cantidad de desarrolladores a nivel mundial creen nuevas herramientas y librerías de código, abriendo el abanico de posibilidades para quien desee experimentar.

Es importante destacar que durante el desarrollo de esta tesis se participó en la comunidad científica logrando las siguientes publicaciones:

- Criptografía Post Cuántica, presentado en el 19no Edición del Workshop de Investigadores en Ciencias de la Computación WICC (Córdoba y Méndez-Garabetti, 2017b).
- Aplicación de criptografía post cuántica en entornos de producción basados en open source, presentado en el 9no Encuentro de Investiga-

dores y Docentes de Ingeniería EnIDI (Córdoba y Méndez-Garabetti, 2017a).

- Criptografía Post-Cuántica Integrada en SSL/TLS Y HTTPS, presentado en la 21ra Edición del Workshop de Investigadores en Ciencias de la Computación WICC (Córdoba y Méndez-Garabetti, 2019).
- Servicio HTTPS con intercambio de claves resistente a ataques cuánticos, presentado en el 7mo Congreso Nacional de Ingeniería Informática Sistemas de Información CONAIISI 2019 (Córdoba et al., 2019).
- Implementación de criptografía post-cuántica NTRU en servicios HTTPS, presentado en el 8vo Congreso Nacional de Ingeniería Informática/Sistemas de Información CONAIISI 2020 (Córdoba et al., 2020).
- Computadoras cuánticas y el futuro de la criptografía, conferencia magistral presentada en el XI Encuentro de Investigadores y Docentes de Ingeniería ENIDI (Córdoba, 2021a).
- Criptografía Post-Cuántica Integrada en HTTPS, conferencia en el Main Track de la Ekoparty Security Conference 2021 (Córdoba, 2021b).

XII TRABAJO FUTURO

Durante la realización del presente trabajo se mantuvo contacto con el equipo de desarrollo y soporte de wolfSSL, quienes brindaron todo el apoyo técnico necesario para poder concretar la integración de su suite SSL/TLS con criptografía post-cuántica en Apache. Además, uno de los programadores y científicos de computación que integran el equipo de wolfSSL, Daniel Stenberg, es también creador del proyecto cURL, una de las herramientas utilizadas en esta tesis. Esto da cuenta del estrecho vínculo que existe entre ambos proyectos, lo cual podría facilitar futuras colaboraciones en la incorporación y prueba de librerías de cifrado post-cuántico en wolfSSL y su integración con cURL.

A continuación se listan algunos tópicos de investigación que se analizarán y desarrollarán en trabajos posteriores a esta tesis:

- Analizar y realizar experimentos de integración de wolfSSL con liboqs para lograr servicios de VPN y HTTPS post-cuánticos basado en algoritmos asimétricos candidatos de la 3ra ronda de estandarización del NIST.
- Analizar y realizar pruebas de concepto de implementaciones de otros protocolos de uso común, tales como SSH o IPSec, con librerías de algoritmos resistentes a ataques cuánticos, y determinar la viabilidad de su uso en entornos experimentales.
- Realizar pruebas de integración de criptografía de clave secreta con la herramienta de VPN Wireguard para verificar su viabilidad.
- Realizar pruebas de integración de algoritmos post-cuánticos en la negociación TLS de la red TOR, y de esta manera analizar su viabilidad y rendimiento.

- Desarrollar librerías de intercambio de claves criptográficas basado en la sincronización de redes neuronales artificiales. Cabe mencionar que el autor de esta tesis se encuentra colaborando en un proyecto de investigación al respecto.
- Evaluar la posibilidad de integrar bibliotecas de cifrado post-cuántico en implementaciones de otros servicios, tales como SMTP, IMAP o LDAP.
- Desarrollar una distribución GNU/Linux que incorpore librerías e implementaciones de criptografía post-cuántica con la intención de facilitar pruebas de validez y rendimiento (*benchmarking*) por parte de terceros interesados.

Índice de figuras

Índice de tablas

Índice de algoritmos

Bibliografía

Alkim, E., Avanzi, R., Bos, J., Ducas, L., de la Piedra, A., Pöppelmann, T., Schwabe, P., y Stebila, D. (2017). NewHope. https://www.newhopecrypto.org/.

Alkim, E., Ducas, L., Pöppelmann, T., y Schwabe, P. (2016). Post-quantum key exchange a new hope. página 18.

Amagicom AB (2017). Introducing a post-quantum VPN, Mullvad's strategy for a future problem. https://www.mullvad.net/en/blog/2017/12/8/introducing-post-quantum-vpn-mullvads-strategy-future-problem/.

Bahajji, Z. A. e Illyes, G. (2014). HTTPS as a ranking signal. https://webmasters.googleblog.com/2014/08/https-as-ranking-signal.html.

Baktu, T. (2017). NTRU: Quantum-Resistant High Performance Cryptography. https://tbuktu.github.io/ntru/.

Barreto, P. S., Gueron, S., Gueneysu, T., Misoczki, R., Persichetti, E., Sendrier, N., y Tillich, J.-P. (2017). Cake: Code-based algorithm for key encapsulation. En *IMA International Conference on Cryptography and Coding*, páginas 207–226. Springer.

Becker, G. (2008). Merkle signature schemes, merkle trees and their cryptanalysis. *Ruhr-University Bochum, Tech. Rep*.

Bennett, C. H., Bernstein, E., Brassard, G., y Vazirani, U. (1997). Strengths and weaknesses of quantum computing. *SIAM journal on Computing*, 26(5):1510–1523.

Bennett, C. H., Brassard, G., y Mermin, N. D. (1992). Quantum cryptography without Bells theorem. *Physical Review Letters*, 68(5):557–559.

Bernstein, D. J., Brumley, B. B., Chen, M.-S., y Tuveri, N. (2021). OpenSSLNTRU: Faster post-quantum TLS key exchange. *arXiv preprint*

arXiv:2106.08759.

Bernstein, D. J., Buchmann, J., y Dahmen, E. (2009). *Post Quantum Cryptography*. Springer, Berlin, 1ra edition.

Bernstein, D. J. y Lange, T. (2008). Attacking and Defending the McEliece cryptosystem. En *PQCrypto 2008*.

Bernstein, D. J. y Lange, T. (2017). Post-quantum cryptography-dealing with the fallout of physics success. *IACR Cryptology ePrint Archive*, 2017:314.

Bernstein, D. J. y Lange, T. (2020). Post-quantum cryptography Conferences. `https://pqcrypto.org/conferences.html`.

Biryukov, A., Dunkelman, O., Keller, N., Khovratovich, D., y Shamir, A. (2010). Key Recovery Attacks of Practical Complexity on AES-256 Variants with up to 10 Rounds. En Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., y Gilbert, H., editores, *Advances in Cryptology EUROCRYPT 2010*, volumen 6110, páginas 299–319. Springer Berlin Heidelberg, Berlin, Heidelberg. Series Title: Lecture Notes in Computer Science.

Bleichenbacher, D. (1998). Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. En Goos, G., Hartmanis, J., van Leeuwen, J., y Krawczyk, H., editores, *Advances in Cryptology CRYPTO '98*, volumen 1462, páginas 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg. Series Title: Lecture Notes in Computer Science.

Böck, H., Somorovsky, J., y Young, C. (2018). Return of Bleichenbacher's oracle threat (robot. En *27th USENIX Security Symposium (USENIX Security 18)*, páginas 817–849, Baltimore, MD. USENIX Association.

Bos, J., Costello, C., Ducas, L., Mironov, I., Naehrig, M., Nikolaenko, V., Raghunathan, A., y Stebila, D. (2016). Frodo: Take off the Ring! Practical,

Quantum-Secure Key Exchange from LWE. En *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, páginas 1006–1018, Vienna, Austria. ACM Press.

Braithwaite, M. (2016). Experimenting with post-quantum cryptography.(2016). *Google Security Post*.

Córdoba, D. (2021a). Computadoras cunticas y el futuro de la criptografa. https://www.researchgate.net/publication/358594242_Computadoras_cuanticas_y_el_futuro_de_la_criptografia.

Córdoba, D. (2021b). Criptografa post-cuntica integrada en HTTPS. https://www.researchgate.net/publication/358594135_Criptografia_post-cuantica_integrada_en_HTTPS.

Córdoba, D. (2022). Tesis MTI - Diego Córdoba - Criptografía Post-Cuántica aplicada en entornos de producción open source - Repositorio Git. https://gitlab.com/d1cor/mti_thesis.

Córdoba, D. y Méndez-Garabetti, M. (2017a). Aplicacin de criptografa post cuntica en entornos de produccin basados en open source. *9no Encuentro de Investigadores y Docentes de Ingeniería EnIDI*, página 3.

Córdoba, D. y Méndez-Garabetti, M. (2017b). CRIPTOGRAFÍA POST CUÁNTICA. *19no Edición del Workshop de Investigadores en Ciencias de la Computación WICC*, página 5.

Córdoba, D. y Méndez-Garabetti, M. (2019). CRIPTOGRAFÍA POST-CUÁNTICA INTEGRADA EN SSL/TLS Y HTTPS. *21er Edición del Workshop de Investigadores en Ciencias de la Computación WICC*, página 5.

Córdoba, D., Méndez-Garabetti, M., y Guibout, J. G. (2019). Servicio HTTPS con intercambio de claves resistente a ataques cunticos. *7mo Congreso Nacional de Ingeniería Informática Sistemas de Información CONAIISI 2019*, página 8.

Córdoba, D., Méndez-Garabetti, M., y Guibout, J. G. (2020). Implementacin de criptografa post-cuntica NTRU en servicios HTTPS. *8vo Congreso Nacional de Ingeniería Informática Sistemas de Información CONAIISI 2020*, página 11.

Chen, M.-S., Hülsing, A., Rijneveld, J., Samardjiska, S., y Schwabe, P. (2020). MQDSS post-quantum signature. http://mqdss.org/.

Chromium Blog (2018). A secure web is here to stay. https://blog.chromium.org/2018/02/a-secure-web-is-here-to-stay.html.

Chu, J. (2016). The beginning of the end for encryption schemes? Publisher: MIT News".[Retrieved October 10, 2017], http://news. mi t. ed u/20.

Cimpanu, C. (2017). Firefox Prepares to Mark All HTTP Sites Not Secure After HTTPS Adoption Rises. https://bit.ly/3c4NDQ4/.

Clement, J. (2020). Coronavirus: Impact on online usage in the US-Statistics & facts. *Hamburg: Statista*.

Comer, D. E. (2014). *Internetworking with tcp/ip*. Pearson.

Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, W. T., et al. (2008). Internet x. 509 public key infrastructure certificate and certificate revocation list (crl) profile. *RFC*, 5280:1–151.

Costello, C., Jao, D., Naehrig, M., Renes, J., y Urbanik, D. (2016). Efficient compression of SIDH public keys. https://eprint.iacr.org/2016/963.

Crispin, M. (2003). INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. https://tools.ietf.org/html/rfc3501.

Daemen, J. y Rijmen, V. (1999). AES Proposal: Rijndael. http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf.

Dierks, T. y Rescorla, E. (2008). The Transport Layer Security (TLS) Protocol Version 1.2. https://tools.ietf.org/html/rfc5246.

Ding, J. y Schmidt, D. (2005). Rainbow, a new multivariable polynomial signature scheme. En *International conference on applied cryptography and network security*, páginas 164–175. tex.organization: Springer.

Docker Inc. (2013). Empowering App Development for Developers | Docker. https://www.docker.com/.

Donenfeld, J. A. (2017). WireGuard: Next Generation Kernel Network Tunnel. En *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA. Internet Society.

Donenfeld, J. A. (2020). Protocol & Cryptography - WireGuard. https://www.wireguard.com/protocol/.

Donta, P. K. (2007). Performance analysis of security protocols. Publisher: UNF Digital Commons.

Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., y Stehlé, D. (2018). CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. página 31.

Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., y Stehlé, D. (2019). CRYSTALS-Kyber (version 2.0) Submission to round 2 of the NIST post-quantum project. página 31.

ECRYPT (2008). The eSTREAM portfolio page. https://www.ecrypt.eu.org/stream/project.html.

ETSI, E. T. S. I. (2015). *Quantum Safe Cryptography and Security*. ETSI, France, etsi white paper no. 8 edition.

Falcon (2020). Falcon. https://falcon-sign.info/.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., y Berners-Lee, T. (2014). Hypertext Transfer Protocol – HTTP/1.1 (RFCs 7230-7237). https://tools.ietf.org/html/rfc2616.

Fluhrer, S., Kampanakis, P., McGrew, D., y Smyslov, V. (2020). Mixing Preshared Keys in the Internet Key Exchange Protocol Version 2 (IKEv2) for Post-quantum Security. Technical Report RFC8784, RFC Editor.

Fluhrer, S., McGrew, D., Kampanakis, P., y Smyslov, V. (2017). Postquantum preshared keys for IKEv2. *Internet Engineering Task Force, Internet-Draft draft-ietf-ipsecme-qr-ikev2-08*.

Frankel, S. y Krishnan, S. (2011). IP security (IPsec) and internet key exchange (IKE) document roadmap. *RFC*, 6071:1–63.

Freier, A. y Karlton, P. (2011). The Secure Sockets Layer (SSL) Protocol Version 3.0. `https://tools.ietf.org/html/rfc6101`.

George, T., Li, J., Fournaris, A. P., Zhao, R. K., Sakzad, A., y Steinfeld, R. (2021). Performance evaluation of post-quantum TLS 1.3 on embedded systems. *Cryptology ePrint Archive*.

Gisin, N., Ribordy, G., Tittel, W., y Zbinden, H. (2002). Quantum cryptography. *Reviews of modern physics*, 74(1):145.

Google Inc. (2020). boringssl - Git at Google. `https://boringssl.googlesource.com/boringssl/`.

Grimes, R. A. (2019). *Cryptography apocalypse: Preparing for the day when quantum computing breaks today's crypto*. John Wiley & Sons.

Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. En *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, páginas 212–219.

Hamburg, M. (2019). Post-quantum cryptography proposal: ThreeBears. *NIST PQC Round*, 2:4.

Hartmanis, J. (1982). Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson). *Siam Review*, 24(1):90.

Hermans, J., Vercauteren, F., y Preneel, B. (2010). Speed Records for NTRU. En Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., y Pieprzyk, J., editores, *Topics in Cryptology - CT-RSA 2010*, volumen 5985, páginas 73–88. Springer Berlin Heidelberg, Berlin, Heidelberg. Series Title: Lecture Notes in Computer Science.

Herrera, O. (2019). HerraduraKEx. https://github.com/Caume/HerraduraKEx.

Hoffstein, J., Pipher, J., y Silverman, J. H. (1998). NTRU: A ring-based public key cryptosystem. En *International algorithmic number theory symposium*, páginas 267–288. Springer.

Horwitz, L. (2021). IoT Trends 2021: A Focus on Fundamentals, Not Nice-to-Haves. https://www.iotworldtoday.com/2021/01/07/iot-trends-2021-a-focus-on-fundamentals-not-nice-to-haves/.

Housley, R. (2009). RFC 5652 - Cryptographic Message Syntax (CMS). https://tools.ietf.org/html/rfc5652.

IBM (2021). IBM Key Protect docs. https://cloud.ibm.com/docs/key-protect.

IBM (2022). Introduction to Quantum-safe Cryptography in TLS. https://cloud.ibm.com/docs/key-protect?topic=key-protect-quantum-safe-cryptography-tls-introduction.

ITU-T (2019). X.509:Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks. https://www.itu.int/rec/T-REC-X.509.

Jao, D., Azarderakhsh, R., Campagna, M., Costello, C., De Feo, L., Hess, B., Jalali, A., Koziel, B., LaMacchia, B., Longa, P., Naehrig, M., Pereira, G., Renes, J., Soukharev, V., y Urbanik, D. (2019). SIKE – Supersingular Isogeny Key Encapsulation. https://sike.org.

Javurek, I. M., Turaník, I. M., y Okay, I. M. (2013). Use of Artificial Neural Networks in Cryptography. página 13.

Kalliauer, J. (2017). Double-slit experiment - In Wikipedia. https://en.wikipedia.org/wiki/{Double}-slit_experiment.

Kannwischer, M. J., Rijneveld, J., Schwabe, P., y Stoffelen, K. (2022). PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

Karp, R. M. (1975). On the computational complexity of combinatorial problems. *Networks*, 5(1):45–68.

Kaufman, C. (2005). Internet key exchange (IKEv2) protocol. *RFC*, 4306:1–99.

Kerschbaumer, C., Gaibler, J., Edelstein, A., y van der Merwe, T. (2021). HTTPS-Only: Upgrading all connections to https in web browsers.

Kinzel, W. y Kanter, I. (2002). Neural cryptography. En *Proceedings of the 9th international conference on neural information processing, 2002. ICONIP'02.*, volumen 3, páginas 1351–1354. tex.organization: IEEE.

Klensin, J. (2001). Simple Mail Transfer Protocol. `https://www.ietf.org/rfc/rfc2821.txt`.

Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209.

Koziel, B., Azarderakhsh, R., y Kermani, M. M. (2018). A high-performance and scalable hardware architecture for isogeny-based cryptography. *IEEE Transactions on Computers*, 67(11):1594–1609. Publisher: IEEE.

Koziel, B., Azarderakhsh, R., Kermani, M. M., y Jao, D. (2016). Post-quantum cryptography on FPGA based on isogenies on elliptic curves. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(1):86–99. Publisher: IEEE.

Kratochvil, M. (2018). Codecrypt - The post-quantum cryptography tool. `https://github.com/exaexa/codecrypt`.

LeonHeTheFirst (2016). SweetOnions - Making Onion Routing Great Again. `https://github.com/LeonHeTheFirst/SweetOnions`.

Lucena López, J. M. (2011). *Criptografía y Seguridad En Computadores*. Universidad de Jaén, España, 4ta edition.

Lyubashevsky, V., Peikert, C., y Regev, O. (2012). On ideal lattices and learning with errors over rings.

Madden, N. (2021). Hybrid encryption and the KEM/DEM paradigm Neil Madden. https://neilmadden.blog/2021/01/22/hybrid-encryption-and-the-kem-dem-paradigm/.

MaitreRenard (2020). SweetOnions - Making Onion Routing Great Again. https://github.com/LeonHeTheFirst/SweetOnions.

Manger, J. (2001). A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0. En Goos, G., Hartmanis, J., van Leeuwen, J., y Kilian, J., editores, *Advances in Cryptology CRYPTO 2001*, volumen 2139, páginas 230–238. Springer Berlin Heidelberg, Berlin, Heidelberg. Series Title: Lecture Notes in Computer Science.

Max Roser, H. R. y Ortiz-Ospina, E. (2015). Internet. *Our World in Data*. https://ourworldindata.org/internet.

McEliece, R. J. (1978). A public-key cryptosystem based on algebraic coding theory. *Coding Thv*, 4244:114–116.

McGrew, D., Curcio, M., y Fluhrer, S. (2019). RFC 8554 - Leighton-Micali Hash-Based Signatures. https://tools.ietf.org/html/rfc8554.

Menezes, A. J., Van Oorschot, P. C., y Vanstone, S. A. (2018). Handbook of applied cryptography. páginas 496–497. CRC press.

Michie, D., Spiegelhalter, D. J., y Taylor, C. C. (1994). Machine learning, neural and statistical classification.

Microsoft (2018). Microsoft PQCrypto-VPN 1.0. https://github.com/Microsoft/PQCrypto-VPN/releases/.

Microsoft (2018). Welcome to the PQCrypto-VPN project! https://github.com/Microsoft/PQCrypto-VPN.

Mojžíš, J. (2018). Tinyssh. https://tinyssh.org/.

Möller, B., Duong, T., y Kotowicz, K. (2014). This poodle bites: exploiting the ssl 3.0 fallback. *Security Advisory*, 21:34–58.

Myers, M., Ankney, R., Malpani, A., Galperin, S., y Adams, C. (1999). X. 509 internet public key infrastructure online certificate status protocol-ocsp.

NAP (2019). *Quantum computing: progress and prospects*. National Academies of Sciences, Engineering, and Medicine and others. National Academies Press.

Nginx, Inc. (2018). Nginx. https://nginx.org/.

NIST (2018). Post-Quantum Cryptography Standardization - Post-Quantum Cryptography | CSRC. https://csrc.nist.gov/Projects/post-quantum-cryptography/Post-Quantum-Cryptography-Standardization.

NIST (2020). National Institute of Standards and Technology. https://www.nist.gov/.

OpenSSH (2021). OpenSSH. https://www.openssh.com/.

OpenSSL Software Foundation (2018). OpenSSL Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.

OpenVPN Inc. (2018). OpenVPN - Open Source VPN. https://openvpn.net/.

OQS (2018). GitHub - open-quantum-safe/openssh: Fork of OpenSSH that includes prototype quantum-resistant key exchange and authentication in SSH based on liboqs. https://github.com/open-quantum-safe/openssh.

OQS (2020). OQS-BoringSSL. https://github.com/open-quantum-safe/boringssl.

OQS (2021a). Liboqs. https://github.com/open-quantum-safe/liboqs.

OQS, O. Q. S. (2021b). OQS-OpenSSL_1_1_1 (git). https://github.com/open-quantum-safe/openssl/tree/OQS-OpenSSL_1_1_1-stable.

Parkhomenko, A. (2021). Post-quantum VPN. Understanding quantum computers and installing OpenVPN to protect them against future threats HackMag. https://hackmag.com/security/quantum-vpn/.

Patarin, J. (1996). Hidden fields equations (HFE) and isomorphisms of polynomials (IP): Two new families of asymmetric algorithms. En *International conference on the theory and applications of cryptographic techniques*, páginas 33–48. Springer.

Perlner, R. A. y Cooper, D. A. (2009). *Quantum Resistant Public Key Cryptography: A Survey*. National Institute of Standards and Technology, Maryland, EEUU.

Perrin, T. (2018). The Noise Protocol Framework. página 65.

Peterson, L. L. y Davie, B. S. (2007). *Computer networks: a systems approach*. Elsevier.

Postel, J. y Reynolds, J. (1985). RFC 959 - FILE TRANSFER PROTOCOL (FTP). *File Transfer Protocol (FTP)*, páginas 1–69.

Preskill, J. (2018). Quantum computing in the nisq era and beyond. *Quantum*, 2:79.

Prime, N. (2020). NTRU Prime: Intro. `https://ntruprime.cr.yp.to/`.

Q-Success (2022). Usage Statistics and Market Share of Nginx, May 2022.

Ramió Aguirre, J. (2018). *Curso de Criptografía Aplicada*. Madrid.

Regev, O. (2009). On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40.

Rescorla, E. (1999). RFC 2631 - Diffie-Hellman Key Agreement Method. `https://tools.ietf.org/html/rfc2631`.

Rescorla, E. (2000). Http over tls. Publisher: RFC 2818, May.

Rescorla, E. y Dierks, T. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. Publisher: RFC 8446, DOI 10.17487/RFC8446, August 2018,ą https://www. rfc-editor. org .

Rivest, R. L., Shamir, A., y Adlemar, L. (1987). *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Communications of the ACM, EEUU.

RLabs (2020). RLabs/xs. `https://gogs.blitter.com/RLabs/xs`.

Ronen, E., Gillham, R., Genkin, D., Shamir, A., Wong, D., y Yarom, Y. (2019). The 9 lives of bleichenbacher's CAT: New cache ATtacks on TLS implementations. En *2019 IEEE symposium on security and privacy (SP)*, páginas 435–452. tex.organization: IEEE.

Schmidt, P. (2010). Post-Quantum Cryptography. página 60.

Shor, P. W. (1994). Algorithms for quantum computation: discrete logarithms and factoring. En *Proceedings 35th annual symposium on foundations of computer science*, páginas 124–134. Ieee.

Shor, P. W. (1997). Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *Journal SIAM Journal on Computing*, 26(5):1484–1509.

Singh, V. (2015). A Post-Quantum fork of OpenSSL 1.0.2e containing A Practical Ring-LWE Key Exchange Implementation. `https://github.com/vscrypto/openssl-ringlwe`.

Sosinowicz, J. (2020). Support for wolfSSL in OpenVPN - GitHub. `https://github.com/julek-wolfssl/openvpn/commit/074d60f8161a4ffe37c216e36de7e6d668395305`.

Stallings, W. (2005). *Cryptography and Network Security Principles and Practices*. Prentice Hall, EEUU.

Stallman, R. (2016). FLOSS and FOSS - GNU Project - Free Software Foundation. `https://www.gnu.org/philosophy/floss-and-foss.en.html`.

Stebila, D. y Mosca, M. (2017). *Post-Quantum Key Exchange for the Internet and the Open Quantum Safe Project*. Department of Computing and Software, Mc Master University.

Stebila, D., Mosca, M., Paquin, C., Sikerdis, D., Tamvda, G., Baentsch, M., Hu, A., y Barshteyn, I. (2022). oqs-demos. `https://github.com/open-quantum-safe/oqs-demos`.

Steblia, D., Fluhrer, S., y Gueron, S. (2021). Hybrid key exchange in TLS 1.3. Technical report, Internet-Draft draft-ietf-tls-hybrid-design-03. Internet Engineering Task .

Stehlé, D. y Steinfeld, R. (2013). *Making NTRUEncrypt and NTRUSign as Secure as Standard Worst-Case Problems over Ideal Lattices*. Published: Cryptology ePrint Archive, Report 2013/004.

strongSwan, D. (2015). NTRU. `https://wiki.strongswan.org/projects/strongswan/wiki/NTRU`.

Stypiski, M. y Niemiec, M. (2021). Synchronization of Tree Parity Machines using non-binary input vectors. *arXiv:2104.11105 [cs]*. arXiv: 2104.11105.

Sullivan, N. (2016). Padding oracles and the decline of CBC-mode cipher suites. `https://blog.cloudflare.com/padding-oracles-and-the-decline-of-cbc-mode-ciphersuites/`.

Takagi, T., editor (2016). *Post-Quantum Cryptography*. Springer, Fukuoka, Japón, pqcrypto 2016 edition.

The Apache Software Foundation (2018). Apache http server project. `https://httpd.apache.org/`.

The GnuPG Project (2021). The GNU Privacy Guard. `https://gnupg.org/`.

The TOR Project (2021). Tor Project | Anonymity Online. `https://www.torproject.org/`.

Tujner, Z. (2019). Quantum-safe TOR. página 60.

Vandersypen, L. M., Steffen, M., Breyta, G., Yannoni, C. S., Sherwood, M. H., y Chuang, I. L. (2001). Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414(6866):883–887. Publisher: Nature Publishing Group.

Weisstein, E. W. (2005). RSA-640 Factored. *MathWorld Headline News*.

Whyte, W., Zhang, Z., Fluhrer, S., y Garcia-Morchon, O. (2017a). Quantum-Safe Hybrid (QSH) Ciphersuite for Transport Layer Security (TLS) version 1.2. `https://tools.ietf.org/html/draft-whyte-qsh-tls12-02`.

Whyte, W., Zhang, Z., Fluhrer, S., y Garcia-Morchon, O. (2017b). Quantum-safe hybrid (QSH) key exchange for Transport Layer Security (TLS) version 1.3. Technical report, Internet-Draft draft-whyte-qsh-tls13-06, Internet Engineering Task Force.

wolfSSL (2018). wolfSSL Embedded SSL/TLS Library. https://www.wolfssl.com.

WolfSSL (2019). Quantum Safety and wolfSSL - wolfSSL. https://www.wolfssl.com/quantum-safety-wolfssl.

WolfSSL (2020a). GitHub - wolfSSL/wolfssl. https://github.com/wolfSSL/wolfssl.

WolfSSL (2020b). wolfSSL + Apache httpd - wolfSSL. https://www.wolfssl.com/wolfssl-apache-httpd/.

WolfSSL (2020c). wolfSSL Embedded SSL/TLS Library. https://www.wolfssl.com/.

WolfSSL (2021). *wolfSSL User Manual*.

WolfSSL, Inc. (2021a). Hybrid Post Quantum Groups in TLS 1.3 - wolfSSL. https://www.wolfssl.com/hybrid-post-quantum-groups-tls-1-3/.

WolfSSL, Inc. (2021b). wolfSSL v5.0.0 Release - wolfSSL.

WolfSSL, Inc. (2022a). Post Quantum cURL - wolfSSL. https://www.wolfssl.com/post-quantum-curl/.

WolfSSL, Inc. (2022b). Post-Quantum Goodies in wolfSSL 5.1.1: FALCON - wolfSSL. https://www.wolfssl.com/post-quantum-goodies-wolfssl-5-1-1-falcon/.