Strengthening the SSH Protocol with Hybrid Algorithms for Post-Quantum Security

S. Selviyani

Abstract—**Post-Quantum Cryptography (PQC) refers to cryptographic algorithms designed in anticipation of the advent of quantum computers. As we know, many widely used public key algorithms, such as RSA, DSA, and Diffie-Hellman, are vulnerable to attacks by quantum computers due to their reliance on the difficulty of certain mathematical problems. Consequently, significant efforts have been made recently in research, development, the formation of Working Groups (WGs), and the standardization of PQC to prepare for this eventuality. One notable protocol, SSH, has implemented hybrid algorithms that combine NTRU Prime and elliptic curve cryptography (ECC) in their key-exchange algorithm. In this paper, we will discuss the necessity of migrating to PQC as a proactive measure before quantum computers become a reality, and we will explore the mitigation of the SSH protocol using these hybrid algorithms.**

Index Terms—**Post Quantum Security (PQS), Secure Shell Protocol (SSH)**

I. Introduction

The development of quantum computers is a serious threat to all security enthusiasts because of the ability of quantum computers to break cryptographic algorithms that rely on the difficulty of solving mathematical problems efficiently, which are mostly used in classical cryptography [1]. Even though quantum computers do not exist yet, it does not mean that we do not have to prepare or worry about them now. Recently, there has been a lot of research and development from large technology companies such as IBM, which promise that quantum computers will be available in the near future.

Shor's algorithm for the quantum computer can break RSA and Diffie-Hellman (DH), which are widely used in public key cryptography [2]. The Cryptography Relevant Quantum Computer (CRQC) is capable of breaking these systems because Shor's algorithm allows quantum attackers to solve discrete logarithm and integer factorization problems in polynomial time [3]. This breaks the security fundamentals of public key cryptography like RSA, (EC)DH, and (EC)DSA [4].

In response to this development, the US National Institute of Standards and Technology (NIST) is currently running a standardization process to identify the next generation of quantum-secure cryptographic algorithms for key exchange and authentication [5]. In addition, other organizations such as the Internet Engineering Task Force (IETF) are studying the move to post-quantum cryptography, while the IETF is also actively developing Internet drafts that propose a transition to post-quantum cryptography in current protocols, such as Transport Layer Security (TLS) and Secure Shell (SSH) [4].

We are not sure whether the post-quantum encryption algorithms, for example NTRU, will prove secure over time, because there has not been as many years of active research on it as we have on the popular algorithms which have already been researched since the 1970's such as RSA and DH on finite fields, or the 1990's such as on elliptic curve cryptography. The new generation of quantum-secure algorithms needs to have a lot more cryptographic analysis to ensure that they are secure enough to be used and trusted by stakeholders. The migration from classical cryptography to post-quantum cryptography is not easy, but it is necessary to do it. To help transition to the new quantum-safe algorithms, the use of hybrid algorithms is an elegant solution. By combining classic algorithms that have been proven for decades with cryptanalysis processes like the elliptic curve with a new algorithm for post-quantum cryptography such as NTRU Prime, we can ensure that the combined algorithm is not weaker than currently used algorithms. This is generally a good idea.

The main reason why we have to start with the transition now is the threat called **record-now-decrypt-later**, in which the adversary captures and records the communication between the client and the server in the present and attempts to decrypt it in the future when he gets access to a quantum computer. While the transition urgency is increased due to the record-now-decrypt-later threat, the time required to update several network protocols such as TLS and SSH and implementation also contributes to this urgency [6].

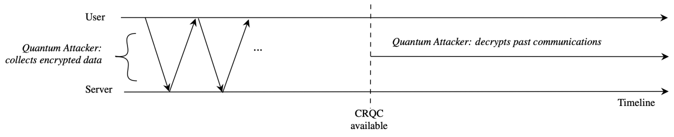

User
Quantum Attacker: collects encrypted data
...
Quantum Attacker: decrypts past communications
Server
Timeline
CRQC available

Fig. 1: Record Now Decrypt Later [6]

Secure Shell (SSH) and Transport Layer Security (TLS, formerly known as Secure Socket Layer or SSL) are protocols which have been commonly used in the last decades. SSH is a program for logging in into and executing commands on a remote computer. SSH is an attractive target for the adversary to record the communication and decrypt it later because it is used to transfer sensitive information such as usernames and passwords on the target machine. Even when keys are used for authentication, system administrators have to enter root passwords and other secrets which they deploy on the server. In addition to remote shell access, SSH is also widely used for file transfer using SCP, SFTP, and rsync, as well as network tunneling for other protocols.

In this paper, the author will focus on the SSH protocol instead of TLS. We will discuss how current IETF RFCs and SSH implementations are strengthening SSH with the use of hybrid algorithms, namely NTRU-Prime and Curve25519. These new implementations are effectively preparing current systems to protect against the *record-now-decrypt-later* attacks.

II. Technical Background

A. Cryptography

Cryptography is the practice and study of techniques for securing information in the presence of adversarial actions. In general, cryptography consists of the creation and analysis of protocols to prevent third parties (i.e., adversaries) from reading or modifying messages sent by the sender to their communications partner [7].

There are two popular types of cryptographic schemes. The first is symmetric cryptography, which uses only a single key to encrypt and decrypt the message. In this case, the party Alice uses the same secret key to encrypt her message, and when she wants to decrypt the message that she encrypted, she will use the same secret key again. If two parties, Alice and Bob, want to use symmetric encryption, they have to share the same secret key. Examples for the symmetric schemes are DES and AES [8].

Secondly, there is asymmetric cryptography (also known as public-key cryptography), where the party Alice has two keys: a secret key and a public key. In contrast to symmetric algorithms, asymmetric algorithms use the two keys for different purposes. The secret key is known only to Alice, whereas the public key can be shared with the public. A famous example of an asymmetric cryptography scheme is the RSA (Rivest-Shamir-Adleman) algorithm. For instance, Alice shares her public key, and anybody, including Bob, can encrypt a message to her. Conversely, only Alice has the matching secret key, which can be used to decrypt the message that Bob sent to her. In addition, many public-key schemes can be used to sign messages with secret keys. The public can verify the signature using the public key.

In the practical implementations of cryptographic protocols, we use hybrid protocols, which means we combine symmetric and asymmetric cryptography. Hybrid protocols have the advantage that they save expensive public-key operations, because symmetric-key cryptography is usually more efficient. Note that these hybrid protocols combine asymmetric and symmetric algorithms. They are not to be confused with the hybrid use of classical and post-quantum algorithms.

In public key cryptography, **key exchange** is a technique allowing Alice and Bob to securely exchange a shared secret key via an insecure communication channel. The key is then used to encrypt and decrypt communication between them. Key exchange works as follows: Alice and Bob each have a pair of keys, one being a public key and the other a private key. With these keys, they can do computations to create a shared secret key. The most common key exchange method used is the Diffie-Hellman key exchange, where both parties use public information and their private keys to independently generate an identical shared secret key. This guarantees that only the two parties can decrypt the messages, even if an adversary intercepts the communication.

For the purpose of easier understanding, the author will not explain the mathematical calculations in this paper. In the Diffie-Hellman key exchange scheme, Bob and Alice generate their secret and public keys to become joint keys and distribute the public key. After receiving the correct copy of Alice and Bob's public keys, both parties can compute a shared secret offline. The shared secret can then be used, for example, as a secret key in asymmetric cryptography schemes [9].

In the key exchange scheme above, Alice gives Bob her public key and vice versa to generate their joint keys. However, how can both parties ensure that the public keys they have obtained are authentic and not the key of an adversary impersonating either Alice or Bob? To prevent that, there is a method called **signature**. In public key cryptography, a signature is used to authenticate the integrity of a message. In the case of key distribution, Alice and Bob can get their public keys signed by a trusted party. This is also called certificate.

B. Quantum Computers

A quantum computer is a computer that uses the principles of quantum mechanics, the physics and behavior of tiny particles like electrons and photons. In contrast to conventional computers which store data as bits (which may assume one of two values, 0 or 1), quantum computers use qubits. A single qubit is a quantum memory unit that can contain a superposition of the two states $|0\rangle$ and $|1\rangle$. This is often interpreted as a condition that is *in between* or *at the same time in both states* [10]. The superpositions are related with the probabilities of one or the other state being observed by physicists. It allows quantum computers to process a lot of information simultaneously and resolve specific complex problems far more efficiently than traditional computers.

The classical cryptographic algorithms are based on difficult mathematical problems with the goal that they will not be broken with a modern machine or smart algorithm in the future. Nevertheless, the creation of Shor's algorithm and Grover's algorithm changed the situation. These algorithms can solve difficult mathematical problems used in classical cryptography. One of the devastating effects of Shor's algorithm is that it breaks the RSA algorithm, where the public key is a public exponent e and a product N of two prime numbers p and q. The secret exponent d and the prime numbers p and q are kept secret. The security of RSA can be broken by computing the factors p, q of N. With the introduction of Shor's algorithm in 1994, there is an efficient quantum algorithm to find the prime factorization of the positive integer [2]. Shor's algorithm uses the ability of quantum computers to do several calculations at once using the qubit superposition. It focuses on finding the *period* of a function, which is the duration of a recurring pattern within a sequence. Before using the quantum element in the process, classical techniques are used to select a random number to show the elements of the huge number being analyzed. The quantum computer uses the Quantum Fourier Transform method to determine the period of a mathematical function associated with the integer being factored. This method allows the quantum computer to identify numerical patterns more efficiently than classical computers. After identifying the period, classical computations are executed to figure out the components of the original number, typically more efficient than the quantum element. A variation of Shor's algorithm can also be used to break discrete-logarithm based systems like DH and ECDH.

There is not only the devastating effect of Shor's algorithm to the public-key cryptography, but also Grover's algorithm, a quantum algorithm which weakens symmetric algorithms like AES.

Name	Function	Pre-Quantum Security Level	Post-Quantum Security Level
Symmetric Cryptography			
AES-128	Symmetric Enc.	128	64 (Grover)
AES-256	Symmetric Enc.	256	128 (Grover)
Salsa20	Symmetric Enc.	256	128 (Grover)
GMAC	MAC	128	128 (No impact)
Poly1305	MAC	128	128 (No impact)
SHA-256	Hash Function	256	128 (Grover)
SHA3-256	Hash Function	256	128 (Grover)
Public-Key Cryptography			
RSA-3072	Encryption	128	Broken (Shor)
RSA-3072	Signature	128	Broken (Shor)
DH-3072	Key Exchange	128	Broken (Shor)
DSA-3072	Signature	128	Broken (Shor)
256-bit ECDH	Key Exchange	128	Broken (Shor)
256-bit ECDH	Signature	128	Broken (Shor)

Fig. 2: Effect of Shor's and Grover algorithms

Grover's algorithm offers a quadratic improvement in the efficiency of searching an unsorted database. When applied to the problem of brute-forcing symmetric cryptographic keys (i.e., exhaustively checking all potential keys), it significantly speeds the process of identifying the correct key.

Grover's algorithm requires only $2^{n/2}$ steps in finding the correct key, compared to the 2^n steps needed to examine all potential keys with a classical computer. Common algorithms like AES can be used with double the key length to strengthen against Grover's algorithm.

The effects of Shor's and Grover's on classical cryptography are shown in Fig. 2. The table shows the threat of quantum computers breaking public-key cryptography and weakening symmetric algorithms with larger key sizes.

The US-based National Institute of Standards and Technology (NIST) published a paper [11] in April 2016 citing various cryptologists who acknowledge the probable impact of quantum computers on the routinely used RSA algorithm, which is expected to be no longer be secure by 2030. As a result, in December 2016, NIST initiated the standardization process by releasing a request for ideas for a quantum-secure system based on public key cryptography. The competition is currently in its third of four rounds, during which some algorithms are eliminated and others are carefully evaluated.

Additionally, in October 2024, NIST updated its website [5] with a third-round submission of the finalists and alternative candidates for public-key encryption, key establishment, and digital signature algorithms.

Type	**Public-Key Encryption / Key Encapsulation**	**Digital Signature**
Lattice	• CRYSTALS-Kyber • NTRU • SABER	• CRYSTALS-Dilithium • FALCON
Code-based	Classic McEliece	
Multivariate		Rainbow

Fig. 3: Finalist Post-Quantum Algorithms

Type	Public-Key Encryption / Key Encapsulation	Digital Signature
Lattice	• FrodoKEM • NTRU Prime	
Code-based	• BIKE • HQC	
Hash-based		• SPHINCS+
Multivariate		• GeMSS
Super Singulars Elliptic Curve Isogeny	• SIKE	
Zero-Knowledge Proofs		• Picnic

Fig. 4: Alternative Finalist Post-Quantum Algorithms

C. Hybrid Algorithms

As we write in the introduction, hybrid algorithms are a good idea to help the transition from traditional cryptography to quantum-secure schemes. The hybrid post-quantum protocols combine post-quantum algorithms with traditional cryptography techniques. A common example is the combination of elliptic curve cryptography (ECC) with post-quantum cryptography such as NTRU Prime. The main reason for choosing the hybrid option is confidence in the new algorithms which have not seen as much cryptoanalytic research. The algorithms are combined such that as long as at least one algorithm is not broken, the construction is considered secure [6].

SSH has a new key exchange option which is using sntrup761 with x25519-sha512. In order to understand it, we try to get the a better overview of Curve25519 (x25519-sha512) and Streamlined NTRU Prime (sntrup761) [12].

Curve25519 is a modern Diffie-Hellman scheme based on an Edwards curve, an alternative to commonly used elliptic curves. (Because of its name, the terms Ed25519 and EdDSA are also commonly used.) Curve25519 provides 128 bits of security (with a 256-bit key size) and is designed for use with the Diffie-Hellman key exchange mechanism. Daniel J. Bernstein has suggested to denote the curve as Curve25519 and the Diffie-Hellman function as x25519 [13].

The OpenSSH team added sntrup761 to improve their x25519-based default in version 8.5, released on March 3, 2021. SNTRUp761 (Streamlined NTRU Prime) is a variation of the NTRU cryptographic algorithm that uses lattice-based encryption to protect information. Choosing appropriate parameters, such as key size 761, provides a balance between security and efficiency.

D. Secure Shell Protocol (SSH)

The Secure Shell protocol was developed by Finnish computer scientist Tatu Ylönen in 1995. OpenSSH is an open source implementation of clients and servers for the SSH protocol, which was first released in 1999 under the OpenBSD project [14]. The idea of SSH is to have some a protocol that encrypt and authenticate all the communication between the client and the server. There are two versions of the SSH protocol. SSH-1, which the original version that Tatu Ylönen developed. Secondly, SSH-2, which is the version that is developed by the OpenBSD project for OpenSSH. This version provides many security improvements and a variety of algorithms to choose from. This version is still used until today.

SSH was originally invented for remotely accessing computers with an interactive shell, hence the name. Before SSH, Telnet was used to remotely access computers. However, Telnet is not secure because it does not feature encryption protecting confidentiality and integrity. So, when the client talks to the server, it can be recorded as plain text (including sensitive information such as passwords and keys) and modified to inject malicious commands.

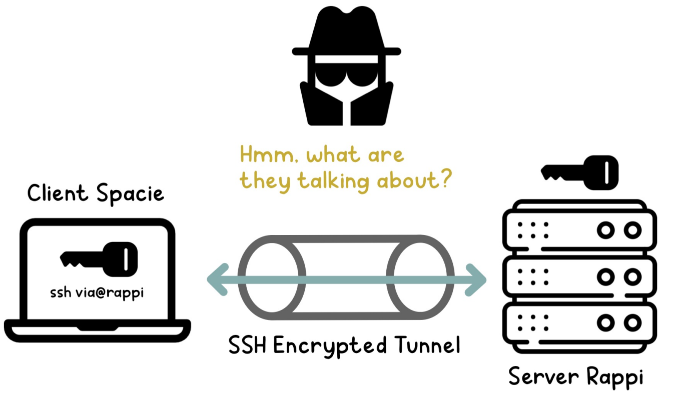


Fig. 5: SSH Tunneling

SSH provides various common cryptographic algorithms such as RSA, AES, ECDSA, SHA etc. for identification of the remote server, authentication of the user, encryption of the messages and integrity of the communication. All modern operating systems support SSH.

The SSH protocol has various features and benefits [15]:

- **Authentication.** SSH verifies the client's and the machine's identities, such as: Who is it? Who do they claim to be? SSH offers two-factor and public-key authentication in addition to password-based authentication.
- **Confidentiality.** By offering end-to-end encryption, only the intended recipients can see the contents. SSH effectively guards against eavesdropping. Additionally, SSH will alert the client when the server has changed its public key (host key fingerprint) to prevent man-in-the-middle attacks.
- **Integrity.** SSH ensures that data transmitted over the network will arrive unmodified using appropriate MACs and block cipher modes.
- **Tunneling.** SSH can establish a secure *tunnel* to allow the transmission of any communication via SSH. This results in similar functionality like proxy servers or VPNs.
- **X11 Forwarding.** SSH protocol can be used on classical Unix machines to forward the X11 protocol in order to use graphical applications on remote machines.

As we can see in Fig. 7, SSH provides a private and a public key. By default, SSH operates on TCP port 22. When we use an SSH server with password authentication on the standard port, a lot of brute-force attacks will occur from adversaries trying to access our server. For that reason, using SSH public-key authentication is strongly recommended. Keys can be generated with `ssh-keygen` tool. In this paper, we will set up the SSH server with sntrup761-x25519 hybrid algorithm to harden our SSH setup.

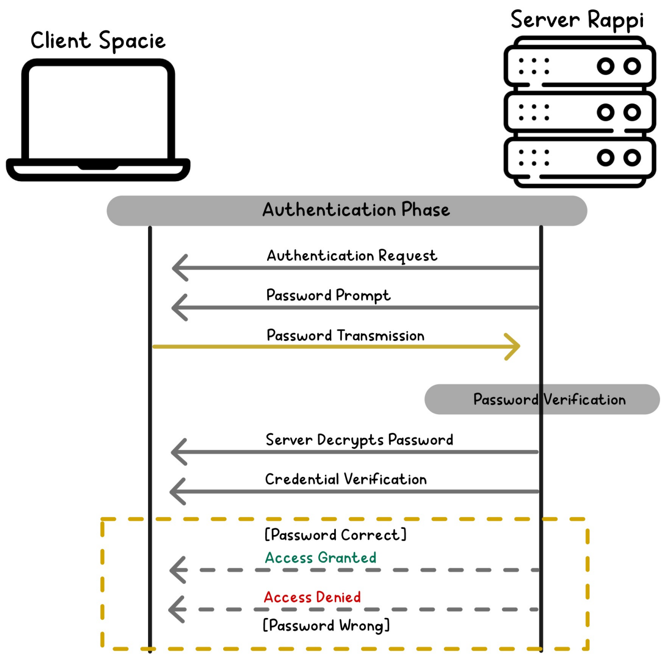


Fig. 6: SSH with Password Authentication

The SSH server is the service program that runs on the device that we wish to connect to. In this experimental setup, we connect with SSH to a Debian-based server that runs on a Raspberry Pi. Since the SSH servers act like our computer's entrance, it is important to make sure that all SSH activity is logged and monitored.

The sequence of how the SSH protocol is depicted in Fig. 6. It depicts SSH with password-based authentication, which is less complex than public-key-based authentication. However, we shall dig deeper into the mechanics of the SSH protocol in the following chapter.

III. Experimental Setup for the Key Exchange in SSH

A. How the SSH work

From the previous chapter, we already drawn the big picture of how SSH works [16]. Crucial for the security of SSH is the

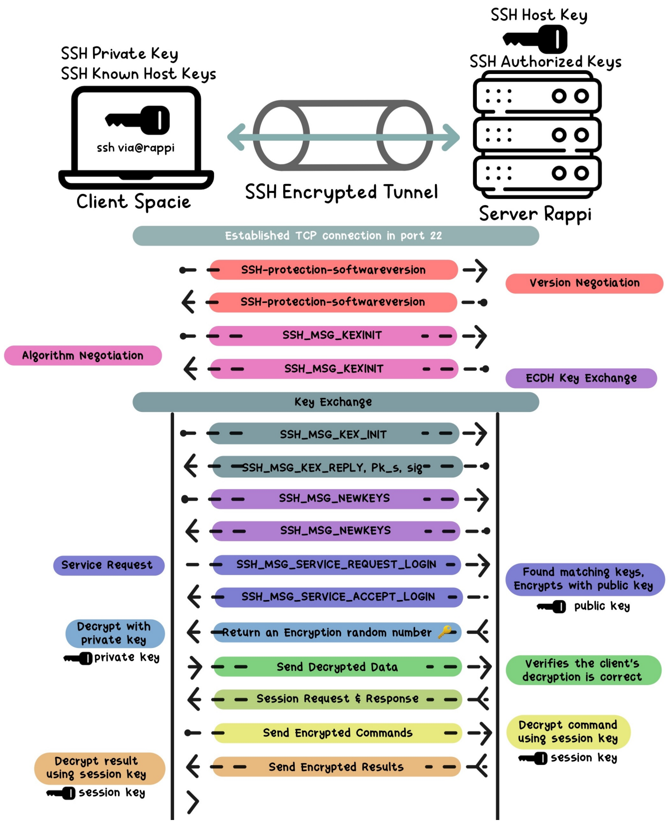
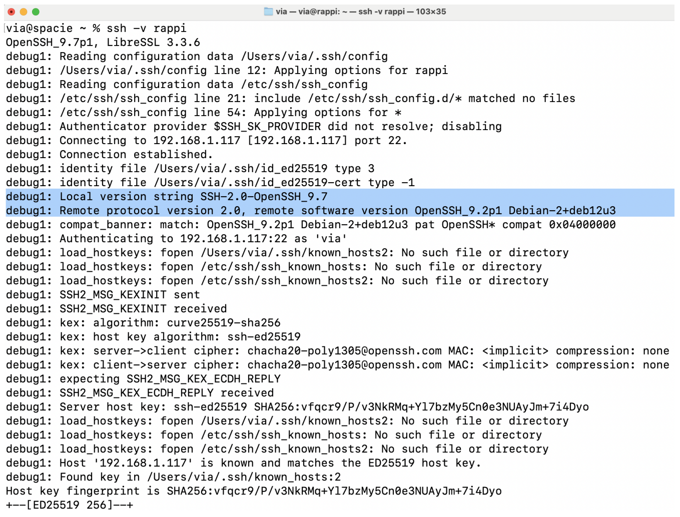
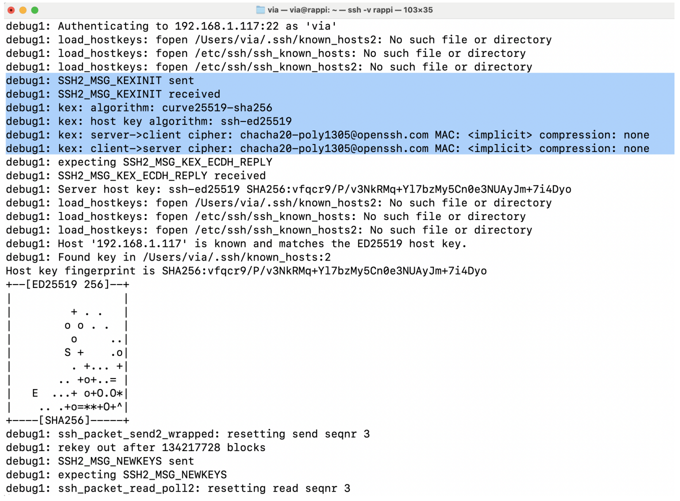


Fig. 7: SSH Handshake Including Key Exchange



```
via — via@rappi: ~ — ssh -v rappi — 103×35
via@spacie ~ % ssh -v rappi
OpenSSH_9.7p1, LibreSSL 3.3.6
debug1: Reading configuration data /Users/via/.ssh/config
debug1: /Users/via/.ssh/config line 12: Applying options for rappi
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: /etc/ssh/ssh_config line 21: include /etc/ssh/ssh_config.d/* matched no files
debug1: /etc/ssh/ssh_config line 54: Applying options for *
debug1: Authenticator provider $SSH_SK_PROVIDER did not resolve; disabling
debug1: Connecting to 192.168.1.117 [192.168.1.117] port 22.
debug1: Connection established.
debug1: identity file /Users/via/.ssh/id_ed25519 type 3
debug1: identity file /Users/via/.ssh/id_ed25519-cert type -1
debug1: Local version string SSH-2.0-OpenSSH_9.7
debug1: Remote protocol version 2.0, remote software version OpenSSH_9.2p1 Debian-2+deb12u3
debug1: compat_banner: match: OpenSSH_9.2p1 Debian-2+deb12u3 pat OpenSSH* compat 0x04000000
debug1: Authenticating to 192.168.1.117:22 as 'via'
debug1: load_hostkeys: fopen /Users/via/.ssh/known_hosts2: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts2: No such file or directory
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
debug1: kex: algorithm: curve25519-sha256
debug1: kex: host key algorithm: ssh-ed25519
debug1: kex: server->client cipher: chacha20-poly1305@openssh.com MAC: <implicit> compression: none
debug1: kex: client->server cipher: chacha20-poly1305@openssh.com MAC: <implicit> compression: none
debug1: expecting SSH2_MSG_KEX_ECDH_REPLY
debug1: SSH2_MSG_KEX_ECDH_REPLY received
debug1: Server host key: ssh-ed25519 SHA256:vfqcr9/P/v3NkRMq+Yl7bzMy5Cn0e3NUAyJm+7i4Dyo
debug1: load_hostkeys: fopen /Users/via/.ssh/known_hosts2: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts2: No such file or directory
debug1: Host '192.168.1.117' is known and matches the ED25519 host key.
debug1: Found key in /Users/via/.ssh/known_hosts:2
Host key fingerprint is SHA256:vfqcr9/P/v3NkRMq+Yl7bzMy5Cn0e3NUAyJm+7i4Dyo
+--[ED25519 256]--+
```

Fig. 8: Protocol Version Establishment



```
via — via@rappi: ~ — ssh -v rappi — 103×35
debug1: Authenticating to 192.168.1.117:22 as 'via'
debug1: load_hostkeys: fopen /Users/via/.ssh/known_hosts2: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts2: No such file or directory
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
debug1: kex: algorithm: curve25519-sha256
debug1: kex: host key algorithm: ssh-ed25519
debug1: kex: server->client cipher: chacha20-poly1305@openssh.com MAC: <implicit> compression: none
debug1: kex: client->server cipher: chacha20-poly1305@openssh.com MAC: <implicit> compression: none
debug1: expecting SSH2_MSG_KEX_ECDH_REPLY
debug1: SSH2_MSG_KEX_ECDH_REPLY received
debug1: Server host key: ssh-ed25519 SHA256:vfqcr9/P/v3NkRMq+Yl7bzMy5Cn0e3NUAyJm+7i4Dyo
debug1: load_hostkeys: fopen /Users/via/.ssh/known_hosts2: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts2: No such file or directory
debug1: Host '192.168.1.117' is known and matches the ED25519 host key.
debug1: Found key in /Users/via/.ssh/known_hosts:2
Host key fingerprint is SHA256:vfqcr9/P/v3NkRMq+Yl7bzMy5Cn0e3NUAyJm+7i4Dyo
+--[ED25519 256]--+
|                 |
|         + . .   |
|        o o . .  |
|         o     ..|
|        S +    .o|
|         . +... +|
|       .. +o+..= |
|   E  ...+ o+O.O*|
|    .. .+o=**+O+^|
+----[SHA256]-----+
debug1: ssh_packet_send2_wrapped: resetting send seqnr 3
debug1: rekey out after 134217728 blocks
debug1: SSH2_MSG_NEWKEYS sent
debug1: expecting SSH2_MSG_NEWKEYS
debug1: ssh_packet_read_poll2: resetting read seqnr 3
```

Fig. 9: Algorithm Negotiation

key exchange process. In this chapter we will go through on each step of the SSH protocol to understand the key exchange.

The **first step** is the initial connection, in which the client attempts to connect to a server by opening a TCP connection over port 22.

The **second step** (highlighted in Fig 8) is the version negotiation between the client and server. By deciding which version of the SSH protocol to use for this session, this step makes sure that both sides are communicating in the same language. Since SSH-1 is no longer secure, they should agree to use SSH-2.

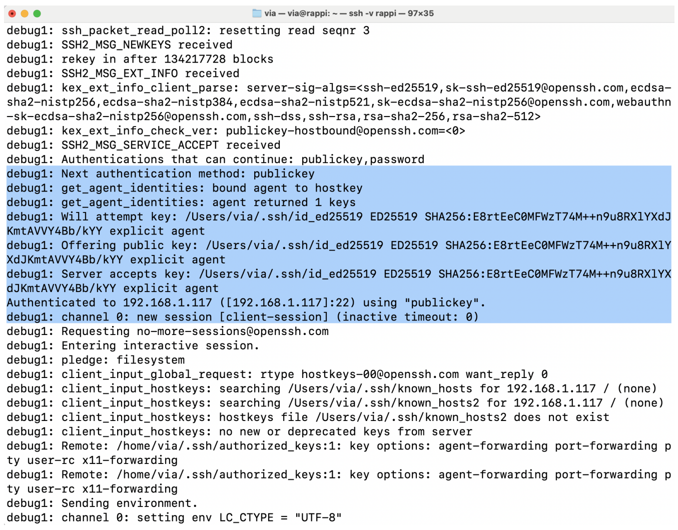
Once the SSH version is established, the **third step** is algorithm negotiation. The client and server determine the cryptographic algorithms for key exchange, encryption, and integrity verification. In this phase, the client request which cryptographic algorithms options that the server supports. In our experimental setup, we want SSH to choose `sntrup761x25519-sha512@openssh.com`.

As the preliminary phase is done, we reach the central step of the protocol: the key exchange. Until recently, the preferred key exchange algorithm in SSH was `curve25519-sha256`. In this process, the client and the server generate ephemeral Curve25519 key pairs and exchange their ephemeral public keys to create a dynamic session key for the encryption. Additionally, the server signs the key exchange message with his long-term public key (host key) to authenticate the server identity [17].

They use the ephemeral key in order to provide **forward secrecy**. This means that even if the long-term keys are leaking in the future, past session data cannot be decrypted. This protects against passive man-in-the-middle attackers who record the messages and gets hand of long-term keys later. After the key exchange, symmetric encryption is applied using this shared secret key to encrypt all of the data sent over the SSH session.

After the key exchange process done, both sides send a trigger message named `SSH_MSG_NEWKEYS` to agree for using the new keys for all coming communication. Now, the client sends a login request to the server: `SSH_MSG_SERVICE_REQUEST_LOGIN`. The server then authenticates the client based on password or user public key. For the public-key authentication, the server looks for a matching public key in its `authorized_keys` file. When a matching key is found, the server encrypts a random integer using the client's public key and returns it to the client. The client then decrypts this random number with his private key, returning it to the server. This way, client proves its identity. This challenge guarantees that the client processes the matched private key.

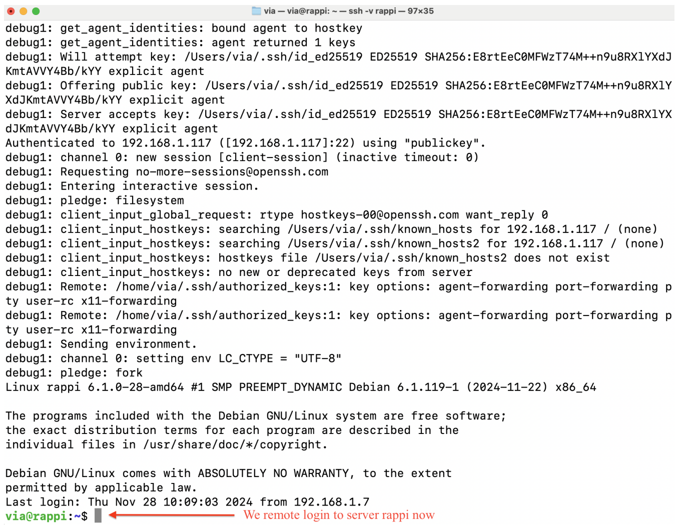
In the next steps, the server and the client agree on the applications, for example remote shell or file transfer. As this part is independent from the key exchange, we will not go into greater detail.



```
debug1: ssh_packet_read_poll2: resetting read seqnr 3
debug1: SSH2_MSG_NEWKEYS received
debug1: rekey in after 134217728 blocks
debug1: SSH2_MSG_EXT_INFO received
debug1: kex_ext_info_client_parse: server-sig-algs=<ssh-ed25519,sk-ssh-ed25519@openssh.com,ecdsa-
sha2-nistp256,ecdsa-sha2-nistp384,ecdsa-sha2-nistp521,sk-ecdsa-sha2-nistp256@openssh.com,webauthn
-sk-ecdsa-sha2-nistp256@openssh.com,ssh-dss,ssh-rsa,rsa-sha2-256,rsa-sha2-512>
debug1: kex_ext_info_check_ver: publickey-hostbound@openssh.com=<0>
debug1: SSH2_MSG_SERVICE_ACCEPT received
debug1: Authentications that can continue: publickey,password
debug1: Next authentication method: publickey
debug1: get_agent_identities: bound agent to hostkey
debug1: get_agent_identities: agent returned 1 keys
debug1: Will attempt key: /Users/via/.ssh/id_ed25519 ED25519 SHA256:E8rtEeC0MFWzT74M++n9u8RXlYXdJ
KmtAVVY4Bb/kYY explicit agent
debug1: Offering public key: /Users/via/.ssh/id_ed25519 ED25519 SHA256:E8rtEeC0MFWzT74M++n9u8RXlY
XdJKmtAVVY4Bb/kYY explicit agent
debug1: Server accepts key: /Users/via/.ssh/id_ed25519 ED25519 SHA256:E8rtEeC0MFWzT74M++n9u8RXlYX
dJKmtAVVY4Bb/kYY explicit agent
Authenticated to 192.168.1.117 ([192.168.1.117]:22) using "publickey".
debug1: channel 0: new session [client-session] (inactive timeout: 0)
debug1: Requesting no-more-sessions@openssh.com
debug1: Entering interactive session.
debug1: pledge: filesystem
debug1: client_input_global_request: rtype hostkeys-00@openssh.com want_reply 0
debug1: client_input_hostkeys: searching /Users/via/.ssh/known_hosts for 192.168.1.117 / (none)
debug1: client_input_hostkeys: searching /Users/via/.ssh/known_hosts2 for 192.168.1.117 / (none)
debug1: client_input_hostkeys: hostkeys file /Users/via/.ssh/known_hosts2 does not exist
debug1: client_input_hostkeys: no new or deprecated keys from server
debug1: Remote: /home/via/.ssh/authorized_keys:1: key options: agent-forwarding port-forwarding p
ty user-rc x11-forwarding
debug1: Remote: /home/via/.ssh/authorized_keys:1: key options: agent-forwarding port-forwarding p
ty user-rc x11-forwarding
debug1: Sending environment.
debug1: channel 0: setting env LC_CTYPE = "UTF-8"
```

Fig. 10: Public-Key Authentication

Once authentication is completed, the SSH session is fully constructed, from this point into the future, all commands transmitted from the client to the server are encrypted with the session key. The server executes these commands, encrypts the results with the same session key, and returns them to the client. The client then decrypts this result with the session key, this encrypted back and forth occurs during the SSH session.


via — via@rappi: ~ — ssh -v rappi — 97×35
We remote login to server rappi now

```
debug1: get_agent_identities: bound agent to hostkey
debug1: get_agent_identities: agent returned 1 keys
debug1: Will attempt key: /Users/via/.ssh/id_ed25519 ED25519 SHA256:E8rtEeC0MFWzT74M++n9u8RXlYXdJ
KmtAVVY4Bb/kYY explicit agent
debug1: Offering public key: /Users/via/.ssh/id_ed25519 ED25519 SHA256:E8rtEeC0MFWzT74M++n9u8RXlY
XdJKmtAVVY4Bb/kYY explicit agent
debug1: Server accepts key: /Users/via/.ssh/id_ed25519 ED25519 SHA256:E8rtEeC0MFWzT74M++n9u8RXlYX
dJKmtAVVY4Bb/kYY explicit agent
Authenticated to 192.168.1.117 ([192.168.1.117]:22) using "publickey".
debug1: channel 0: new session [client-session] (inactive timeout: 0)
debug1: Requesting no-more-sessions@openssh.com
debug1: Entering interactive session.
debug1: pledge: filesystem
debug1: client_input_global_request: rtype hostkeys-00@openssh.com want_reply 0
debug1: client_input_hostkeys: searching /Users/via/.ssh/known_hosts for 192.168.1.117 / (none)
debug1: client_input_hostkeys: searching /Users/via/.ssh/known_hosts2 for 192.168.1.117 / (none)
debug1: client_input_hostkeys: hostkeys file /Users/via/.ssh/known_hosts2 does not exist
debug1: client_input_hostkeys: no new or deprecated keys from server
debug1: Remote: /home/via/.ssh/authorized_keys:1: key options: agent-forwarding port-forwarding p
ty user-rc x11-forwarding
debug1: Remote: /home/via/.ssh/authorized_keys:1: key options: agent-forwarding port-forwarding p
ty user-rc x11-forwarding
debug1: Sending environment.
debug1: channel 0: setting env LC_CTYPE = "UTF-8"
debug1: pledge: fork
Linux rappi 6.1.0-28-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.119-1 (2024-11-22) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Nov 28 10:09:03 2024 from 192.168.1.7
via@rappi:~$ 
```

Fig. 11: SSH Connection to Rappi

B. Key Exchange Setup with the PQC Algorithm

In Fig. 12 following picture, we can see in detail how SSH works with the PQC key exchange.

Normally, setting up SSH on our computers is relatively easy. We just need to run a command such as `apt install openssh-client` on the desktop and `apt install openssh-server` in the server. Most servers and Linux or Mac computers have it preinstalled. The default configuration is `/etc/ssh/ssh_config` for the client and `/etc/ssh/sshd_config` for the server. These files we will modify to enable PQC key exchange.

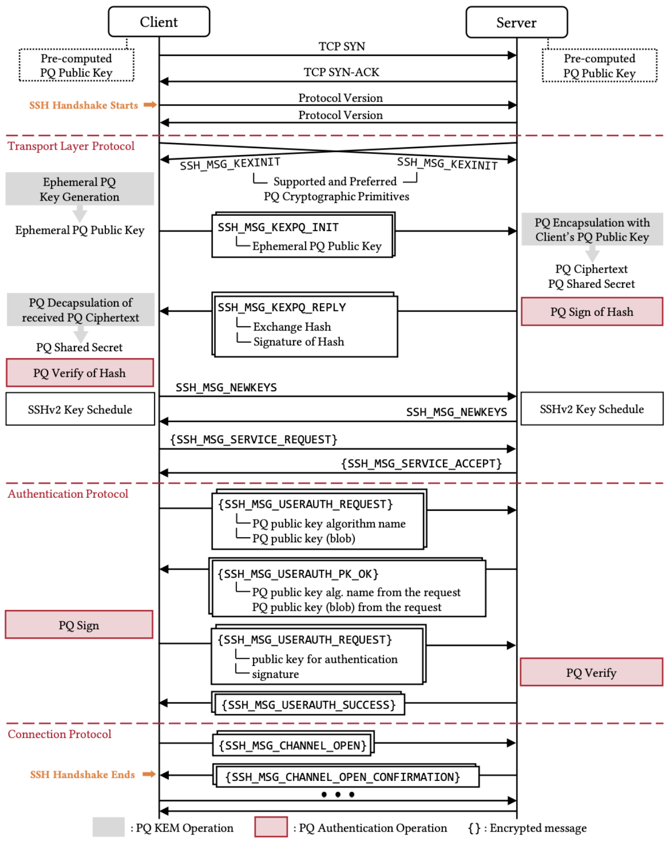


Fig. 12: SSH Protocol with PQC [4]

By default, the public key for authentication is saved in the file `.ssh/id_ed25519.pub`, while the file `.ssh/id_ed26619` is the secret key. Then, we can copy the public key to the remote machine and add it to the `authorized_keys`.

Since the `sshd_config` is from an older SSH installation, the will automatically choose `curve25519-sha256` as the priority in the algorithm list. We want to change the priority of `KexAlgorithms` to `sntrup761x25519-sha512`. The following paragraphs will explain each step for the configuration.

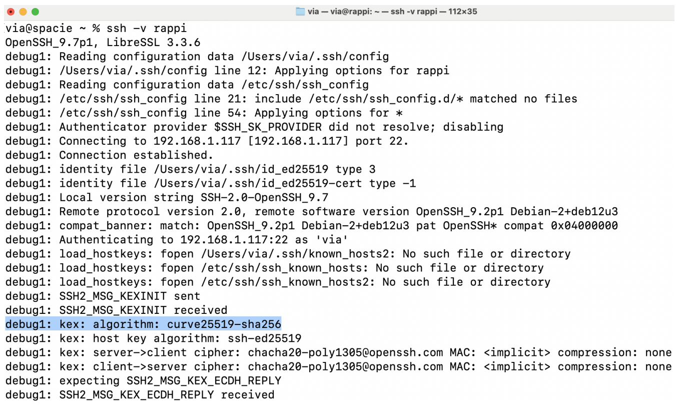
In our experimental setup, we will use the following client machine:

- Hostname: `spacie`
- Chipset: Apple M2 (Macbook Pro)
- Memory: 32 GB
- OS: macOS 14 (Sonoma)

For the server, we use the following machine:

- Hostname: `rappi`
- Chipset: ARM Cortex-A53 (Raspberry Pi)
- Memory: 1 GB
- OS: Debian GNU/Linux 12.8 (Bookworm)

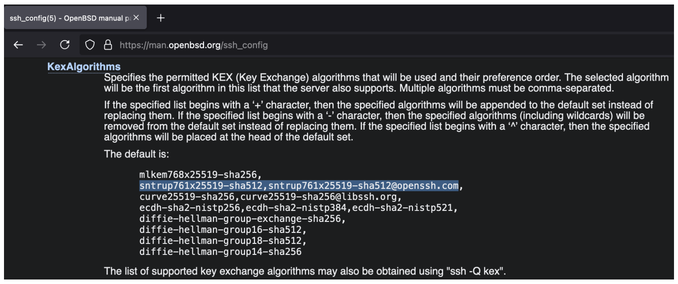
When we run SSH in verbose mode (`ssh -v rappi`), we can see the debugging messages about its progress. We can see in Fig. 13 that `curve25519-sha256` is chosen.



```
via@spacie ~ % ssh -v rappi
OpenSSH_9.7p1, LibreSSL 3.3.6
debug1: Reading configuration data /Users/via/.ssh/config
debug1: /Users/via/.ssh/config line 12: Applying options for rappi
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: /etc/ssh/ssh_config line 21: include /etc/ssh/ssh_config.d/* matched no files
debug1: /etc/ssh/ssh_config line 54: Applying options for *
debug1: Authenticator provider $SSH_SK_PROVIDER did not resolve; disabling
debug1: Connecting to 192.168.1.117 [192.168.1.117] port 22.
debug1: Connection established.
debug1: identity file /Users/via/.ssh/id_ed25519 type 3
debug1: identity file /Users/via/.ssh/id_ed25519-cert type -1
debug1: Local version string SSH-2.0-OpenSSH_9.7
debug1: Remote protocol version 2.0, remote software version OpenSSH_9.2p1 Debian-2+deb12u3
debug1: compat_banner: match: OpenSSH_9.2p1 Debian-2+deb12u3 pat OpenSSH* compat 0x04000000
debug1: Authenticating to 192.168.1.117:22 as 'via'
debug1: load_hostkeys: fopen /Users/via/.ssh/known_hosts2: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts2: No such file or directory
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
debug1: kex: algorithm: curve25519-sha256
debug1: kex: host key algorithm: ssh-ed25519
debug1: kex: server->client cipher: chacha20-poly1305@openssh.com MAC: <implicit> compression: none
debug1: kex: client->server cipher: chacha20-poly1305@openssh.com MAC: <implicit> compression: none
debug1: expecting SSH2_MSG_KEX_ECDH_REPLY
debug1: SSH2_MSG_KEX_ECDH_REPLY received
```

Fig. 13: SSH with `curve25519-sha256`

We can check for the list of all supported algorithms in the SSH manpage for `ssh_config`. Fig. 14 shows the manpage of the most recent version on the OpenBSD website [18].


ssh_config(5) - OpenBSD manual p
https://man.openbsd.org/ssh_config

KexAlgorithms

Specifies the permitted KEX (Key Exchange) algorithms that will be used and their preference order. The selected algorithm will be the first algorithm in this list that the server also supports. Multiple algorithms must be comma-separated.

If the specified list begins with a '+' character, then the specified algorithms will be appended to the default set instead of replacing them. If the specified list begins with a '-' character, then the specified algorithms (including wildcards) will be removed from the default set instead of replacing them. If the specified list begins with a '^' character, then the specified algorithms will be placed at the head of the default set.

The default is:

```
mlkem768x25519-sha256,
sntrup761x25519-sha512,sntrup761x25519-sha512@openssh.com,
curve25519-sha256,curve25519-sha256@libssh.org,
ecdh-sha2-nistp256,ecdh-sha2-nistp384,ecdh-sha2-nistp521,
diffie-hellman-group-exchange-sha256,
diffie-hellman-group16-sha512,
diffie-hellman-group18-sha512,
diffie-hellman-group14-sha256
```

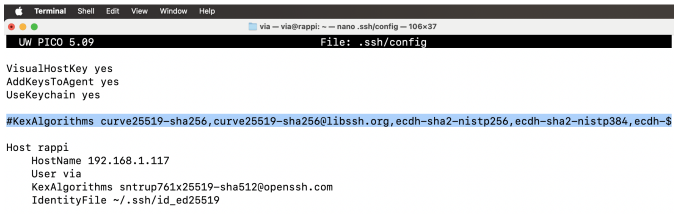
The list of supported key exchange algorithms may also be obtained using "ssh -Q kex".

Fig. 14: Newest SSH Manpage on OpenBSD website

In addition to NTRU Prime, it also shows Kyber768 (`mlkem768x25519-sha256`). However, this algorithm is not supported by our client and server, so we stick to NTRU Prime.

We can also check the server for all configured KexAlgorithms with Nmap (Fig. 15). Nmap is an open source tool for network security with many useful scripts.

As we can see, `rappi` does not have NTRU Prime on the list of KexAlgorithms yet. To strengthen our SSH connection, we will add the PQC algorithm to our `sshd_config`. To achieve this, we log in remotely to the server and modify `sshd_config` as root user by adding `sntrup761x25519-sha512` (Fig. 16). After saving the config, we need to restart SSH service with `systemctl restart ssh`.


Terminal Shell Edit View Window Help
via — via@rappi: ~ — nano .ssh/config — 106×37

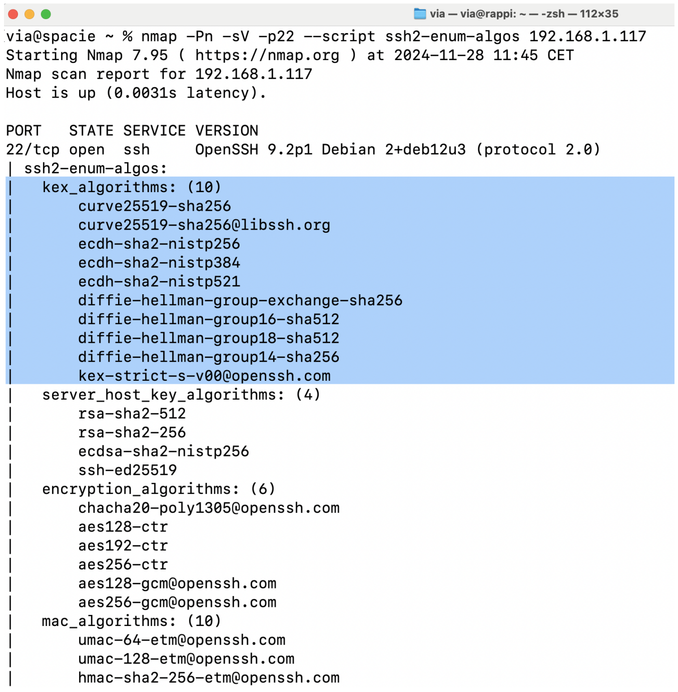
```
  UW PICO 5.09                                                  File: .ssh/config

VisualHostKey yes
AddKeysToAgent yes
UseKeychain yes

#KexAlgorithms curve25519-sha256,curve25519-sha256@libssh.org,ecdh-sha2-nistp256,ecdh-sha2-nistp384,ecdh-$

Host rappi
    HostName 192.168.1.117
    User via
    KexAlgorithms sntrup761x25519-sha512@openssh.com
    IdentityFile ~/.ssh/id_ed25519
```

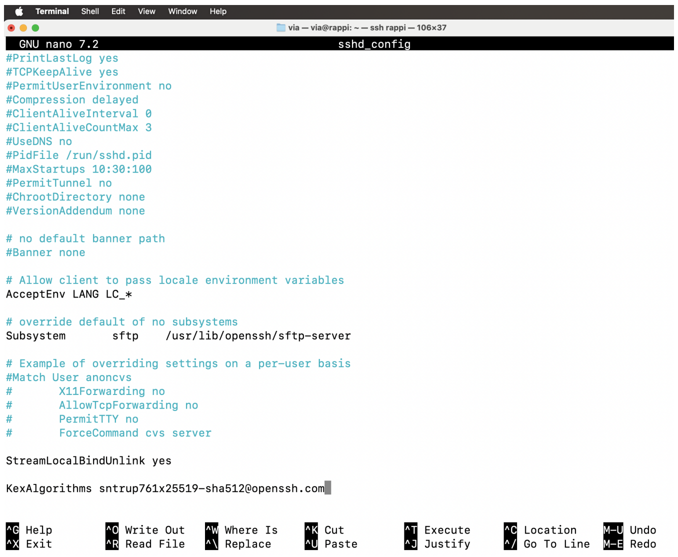
Fig. 17: `ssh_config` on `spacie`



```
via@spacie ~ % nmap -Pn -sV -p22 --script ssh2-enum-algos 192.168.1.117
Starting Nmap 7.95 ( https://nmap.org ) at 2024-11-28 11:45 CET
Nmap scan report for 192.168.1.117
Host is up (0.0031s latency).

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 9.2p1 Debian 2+deb12u3 (protocol 2.0)
| ssh2-enum-algos:
|   kex_algorithms: (10)
|       curve25519-sha256
|       curve25519-sha256@libssh.org
|       ecdh-sha2-nistp256
|       ecdh-sha2-nistp384
|       ecdh-sha2-nistp521
|       diffie-hellman-group-exchange-sha256
|       diffie-hellman-group16-sha512
|       diffie-hellman-group18-sha512
|       diffie-hellman-group14-sha256
|       kex-strict-s-v00@openssh.com
|   server_host_key_algorithms: (4)
|       rsa-sha2-512
|       rsa-sha2-256
|       ecdsa-sha2-nistp256
|       ssh-ed25519
|   encryption_algorithms: (6)
|       chacha20-poly1305@openssh.com
|       aes128-ctr
|       aes192-ctr
|       aes256-ctr
|       aes128-gcm@openssh.com
|       aes256-gcm@openssh.com
|   mac_algorithms: (10)
|       umac-64-etm@openssh.com
|       umac-128-etm@openssh.com
|       hmac-sha2-256-etm@openssh.com
```

Fig. 15: Scanning `rappi` with Nmap


Terminal Shell Edit View Window Help
via — via@rappi: ~ — ssh rappi — 106×37

```
  GNU nano 7.2                                   sshd_config
#PrintLastLog yes
#TCPKeepAlive yes
#PermitUserEnvironment no
#Compression delayed
#ClientAliveInterval 0
#ClientAliveCountMax 3
#UseDNS no
#PidFile /run/sshd.pid
#MaxStartups 10:30:100
#PermitTunnel no
#ChrootDirectory none
#VersionAddendum none

# no default banner path
#Banner none

# Allow client to pass locale environment variables
AcceptEnv LANG LC_*

# override default of no subsystems
Subsystem       sftp    /usr/lib/openssh/sftp-server

# Example of overriding settings on a per-user basis
#Match User anoncvs
#       X11Forwarding no
#       AllowTcpForwarding no
#       PermitTTY no
#       ForceCommand cvs server

StreamLocalBindUnlink yes

KexAlgorithms sntrup761x25519-sha512@openssh.com


^G Help         ^O Write Out    ^W Where Is     ^K Cut          ^T Execute      ^C Location     M-U Undo
^X Exit         ^R Read File    ^\ Replace      ^U Paste        ^J Justify      ^/ Go To Line   M-E Redo
```

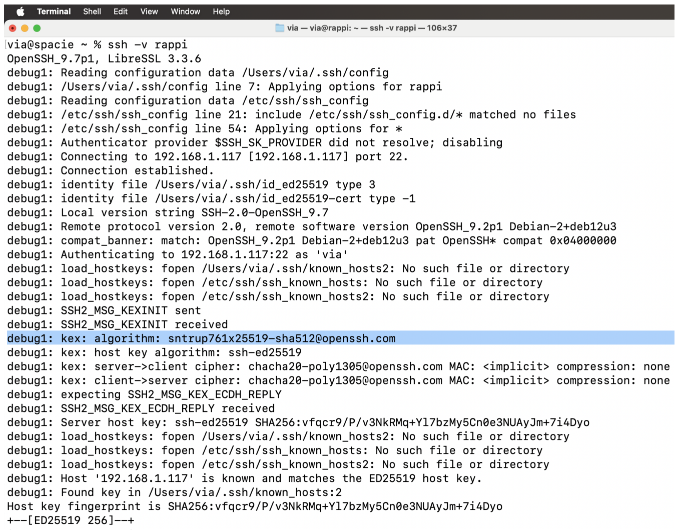
Fig. 16: `sshd_config` on `rappi`

We also want to modify the configuration in `.ssh/config` on the client side. We can force the use of NTRU Prime by setting it as the only option in KexAlgorithms for the specific host (Fig 17).

Finally, we can connect to verify that SSH is now using the PQC algorithm `sntrup761x25519-sha512` in the key exchange process by running it again in verbose mode (Fig. 18).

IV. Conclusion

In our paper, we have seen how the SSH protocol works, making the details visible in verbose mode. SSH has a relatively tricky protocol, but it provides us with security for



```
via@spacie ~ % ssh -v rappi
OpenSSH_9.7p1, LibreSSL 3.3.6
debug1: Reading configuration data /Users/via/.ssh/config
debug1: /Users/via/.ssh/config line 7: Applying options for rappi
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: /etc/ssh/ssh_config line 21: include /etc/ssh/ssh_config.d/* matched no files
debug1: /etc/ssh/ssh_config line 54: Applying options for *
debug1: Authenticator provider $SSH_SK_PROVIDER did not resolve; disabling
debug1: Connecting to 192.168.1.117 [192.168.1.117] port 22.
debug1: Connection established.
debug1: identity file /Users/via/.ssh/id_ed25519 type 3
debug1: identity file /Users/via/.ssh/id_ed25519-cert type -1
debug1: Local version string SSH-2.0-OpenSSH_9.7
debug1: Remote protocol version 2.0, remote software version OpenSSH_9.2p1 Debian-2+deb12u3
debug1: compat_banner: match: OpenSSH_9.2p1 Debian-2+deb12u3 pat OpenSSH* compat 0x04000000
debug1: Authenticating to 192.168.1.117:22 as 'via'
debug1: load_hostkeys: fopen /Users/via/.ssh/known_hosts2: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts2: No such file or directory
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
debug1: kex: algorithm: sntrup761x25519-sha512@openssh.com
debug1: kex: host key algorithm: ssh-ed25519
debug1: kex: server->client cipher: chacha20-poly1305@openssh.com MAC: <implicit> compression: none
debug1: kex: client->server cipher: chacha20-poly1305@openssh.com MAC: <implicit> compression: none
debug1: expecting SSH2_MSG_KEX_ECDH_REPLY
debug1: SSH2_MSG_KEX_ECDH_REPLY received
debug1: Server host key: ssh-ed25519 SHA256:vfqcr9/P/v3NkRMq+Yl7bzMy5Cn0e3NUAyJm+7i4Dyo
debug1: load_hostkeys: fopen /Users/via/.ssh/known_hosts2: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts: No such file or directory
debug1: load_hostkeys: fopen /etc/ssh/ssh_known_hosts2: No such file or directory
debug1: Host '192.168.1.117' is known and matches the ED25519 host key.
debug1: Found key in /Users/via/.ssh/known_hosts:2
Host key fingerprint is SHA256:vfqcr9/P/v3NkRMq+Yl7bzMy5Cn0e3NUAyJm+7i4Dyo
+--[ED25519 256]--+
```

Fig. 18: SSH with `sntrup761x25519-sha512`

remote access to a server over unprotected networks (including the Internet). Indeed, we can make SSH even stronger by adding post-quantum cryptography algorithms to the Key Exchange process. The hybrid algorithm idea of combining the classical algorithm Curve25519 and the PQC algorithm SNTRUp761 is an important first step in anticipation of the existence of Quantum Computer. As we have discussed, the danger of record-now-decrypt-later attacks is already a relevant threat to be considered today. As we have seen, it is relatively easy with regards to the configuration to strengthen SSH installations against the attack. In conclusion, we recommend using the PCQ key exchange and updating older SSH servers for the advanced security benefits.

References

[1] D. D. Tran, K. Ogata, S. Escobar, S. Akleylek, and A. Otmani, "Formal analysis of post-quantum hybrid key exchange ssh transport layer protocol," *IEEE Access*, vol. 12, 2024, published by VRAIN, Universitat Politècnica de València, Ondokuz Mayıs University, University of Tartu, University of Rouen Normandie. [Online]. Available: https://creativecommons.org/licenses/by-nc-nd/4.0/

[2] D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature*, vol. 549, Sep. 2017.

[3] M. Mosca and M. Piani, "Quantum threat timeline report 2020," *Global Risk Insitute. https://globalriskinstitute. org/publications/quantum-threat-timeline-report-2020*, 2021.

[4] D. Sikeridis, P. Kampanakis, and M. Devetsikiotis, "Assessing the overhead of post-quantum cryptography in tls 1.3 and ssh," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 149–156.

[5] N. I. of Standards and Technology, "Post-quantum cryptography," 2024, accessed: 2024-12-12. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography

[6] A. A. Giron, "Migrating applications to post-quantum cryptography: Beyond algorithm replacement," *SECrypt 2023*, 2023, proceedings of the SECrypt 2023 conference.

[7] W. contributors, "Cryptography," Nov. 2001, wikipedia article. [Online]. Available: https://en.wikipedia.org/wiki/Cryptography

[8] K. Paterson, T. W. Barbosa, and D. Ottenheimer, "Sweet32: Birthday attacks on 64-bit block ciphers in tls and openvpn," 2024, website detailing the Sweet32 attack. [Online]. Available: https://sweet32.info/

[9] W. contributors, "Key exchange," Nov. 2024, wikipedia article. [Online]. Available: https://en.wikipedia.org/wiki/Key_exchange

[10] C. Paar, J. Pelzl, and T. Güneysu, *Understanding Cryptography*, second edition ed. Springer, 2024. [Online]. Available: https://doi.org/10.1007/978-3-662-69007-9

[11] W. contributors, "Nist post-quantum cryptography standardization," Nov. 2024, wikipedia article. [Online]. Available: https://en.wikipedia.org/wiki/NIST_Post-Quantum_Cryptography_Standardization

[12] S. Josefsson, "Secure shell (ssh) key exchange method using hybrid streamlined ntru prime sntrup761 and x25519 with sha-512: sntrup761x25519-sha512," May 2023. [Online]. Available: https://www.ietf.org/archive/id/draft-josefsson-ntruprime-ssh-00.html

[13] D. J. Bernstein, "Curve25519: high-speed elliptic-curve cryptography," 2024, website detailing Curve25519 and its cryptographic applications. [Online]. Available: https://cr.yp.to/ecdh.html

[14] IETF, "Rfc 4253: The secure shell (ssh) transport layer protocol," 2024, iETF RFC document. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4253#section-8

[15] Teleport, "What is ssh (secure shell protocol)? advantages & uses explained — teleport," 2024, article on Teleport. [Online]. Available: https://goteleport.com/ssh/

[16] ByteByteGo, "How ssh really works," Nov. 2024, video on YouTube. [Online]. Available: https://www.youtube.com/watch?v=rlMfRa7vfO8

[17] IACR, "Terrapin attack: Breaking ssh channel integrity by sequence number manipulation (rwc 2024)," Apr. 2024, video on YouTube. [Online]. Available: https://www.youtube.com/watch?v=YoNOEpdQ7N8

[18] O. Project, *ssh_config(5) - OpenSSH Configuration File Manual*, 2024, accessed: 2024-12-12. [Online]. Available: https://man.openbsd.org/ssh_config