An explanation of the modern usage of RSA with SHA-512 in SSH

Colin Tang*

June 4, 2025

1 Introduction

SSH is a commonly used protocol for secure communication. The RSA cryptosystem is one of the cryptosystems that can be used by SSH. We give a detailed description of how SSH signs messages using RSA and SHA-512.

2 Background: RSA cryptosystem

An RSA key pair consists of two primes p, q, along with integers d, e, n satisfying the following properties:

- $n = pq$
- $de \equiv 1 \pmod{\lambda(n)}$, where $\lambda(n)$ is the Carmichael totient function

These two properties imply

$$(m^e)^d \equiv m \pmod{n}$$

for any m coprime to n.

The RSA public key is the pair (e, n), whereas the private key can be taken to be the triple (p, q, d). We refer to e, n as the **exponent** and **modulus**, respectively. Given a plaintext message m, the corresponding ciphertext would be $m^e \pmod{n}$, which can be computed using only the public key; given a ciphertext c, it is decrypted by forming $c^d \pmod{n}$.

The length of n (in bits) is usually 3072, but 2048 and 4096 are also commonly seen. These lengths are referred to as RSA-3072, RSA-2048, RSA-4096 respectively.

Usually we take $e = 65537$.

3 Background: RSA key generation

Running `ssh-keygen -t rsa` generates an RSA-3072 key pair (2048- and 4096-bit keys can be generated using the `-b` flag); the private key is usually stored in `~/.ssh/id_rsa` and the public key is usually stored in `~/.ssh/id_rsa.pub`. Let us examine the format of `id_rsa.pub`, using my own public key as an example.

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDkcxbThishY8XNFwqvoKBC4FIKG3eeKc
KqhqP1dKdlfQHlqmovEMvpTwjO7cQ4L78Zz6KwS/zEqJPslefl+P0hFT62yCbyV2j4FB23
R6qkSExGXJ+f31/IsrC/fCnFq8A9cLisJ7cSW9EwNbDoENbd7DCDFy8YSMu/caD1qaY3Ry
2Zaw5lH7PnXBzPBjeAO0AOtIwy6ZCGgRmGxO9w1Nrjhb7oDFjhC05yeGXii3XE6WwK01H/
edLHNt0sy+Sw4lYkmIgqxy6aaQ7dGTZvWJdIPDlfwa86Cm+FHh89MJ43f1/BusnyWB0okx
t6RngtnTCOJATnxKW2aCZao0l6ovOi7bKy4POUrM9lwyW+qccu3qTgpWwvYBDADUoD4K8a
/iufij0pNNnmaRYJHkH0o69WFkMPjvGIe9Ydljx+8hO5eCwGeckOShd1yhOFepytkrsehz
7RYnYedEjZcxBPvmm6tKM4iv7uda6EBYcgyQcMEkAiHuKdA9l1oOx5T167MZ0= colin@C
OMPUTER-NAME
```

*Carnegie Mellon University, United States of America, cstang@andrew.cmu.edu

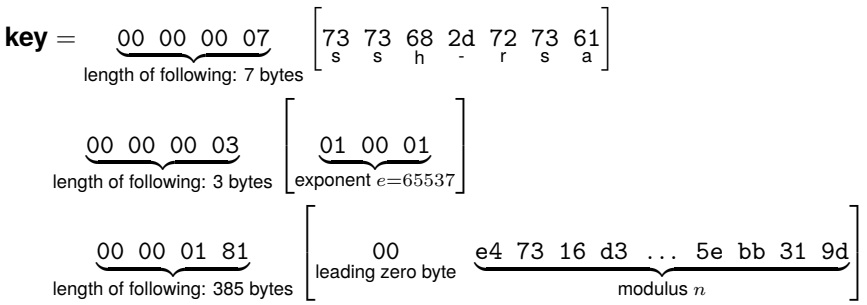
We see that `id_rsa.pub` consists of three parts, separated by spaces:

- The string `ssh-rsa`
- The actual **key** (the 544-character string `AAAA...MZ0=` in the example)
- A comment (`colin@COMPUTER-NAME` in the example)

Note that the **key** `AAAA...MZ0=` is in Base64 format. Let us convert it to hexadecimal:

```
000000077373682d72736100000003010001000018100e47316d3862b2163c5cd170a
afa0a042e0520a1b779e29c2aa86a3f574a7657d01e5aa6a2f10cbe94f08ceedc4382f
bf19cfa2b04bfcc4a893ec95e7e5f8fd21153eb6c826f25768f8141db747aaa4484c46
5c9f9fdf5fc8b2b0bf7c29c5abc03d70b8ac27b7125bd13035b0e810d6ddec3083172f
1848cbbf71a0f5a9a637472d996b0e651fb3e75c1ccf0637803b400eb48c32e9908681
1986c4ef70d4dae385bee80c58e10b4e727865e28b75c4e96c0ad351ff79d2c736dd2c
cbe4b0e2562498882ac72e9a690edd19366f5897483c395fc1af3a0a6f851e1f3d309e
377f5fc1bac9f2581d28931b7a46782d9d308e2404e7c4a5b668265aa3497aa2f3a2ed
b2b2e0f394accf65c325bea9c72edea4e0a56c2f6010c00d4a03e0af1afe2b9f8a3d29
34d9e66916091e41f4a3af5616430f8ef1887bd61d963c7ef213b9782c0679c90e4a17
75ca13857a9cad92bb1e873ed162761e7448d973104fbe69bab4a3388afeee75ae8405
8720c9070c1240221ee29d03d975a0ec794f5ebb319d
```

Now we can see the format of the **key** more clearly.



As we see, the public key consists of three fields, each prepended by a 32-bit integer indicating the length of the respective field (in bytes). The first field is simply the string "`ssh-rsa`". The second field indicates the exponent e, and the third field indicates the modulus n (with a leading zero byte). Note that here, n consists of 384 bytes, hence is a 3072-bit integer; the purpose of the leading zero byte is to ensure that the third field does not have its leftmost bit equal to `1`, since otherwise it would be interpreted as a negative integer.

4 SSH signatures

Running `ssh-keygen -Y sign -f ~/.ssh/id_rsa -n` *namespace* will read data from `stdin`, sign it using the private key `~/.ssh/id_rsa`, and output the signature to `stdout` (here *namespace* is an arbitrary string). Let us see what happens with an example.

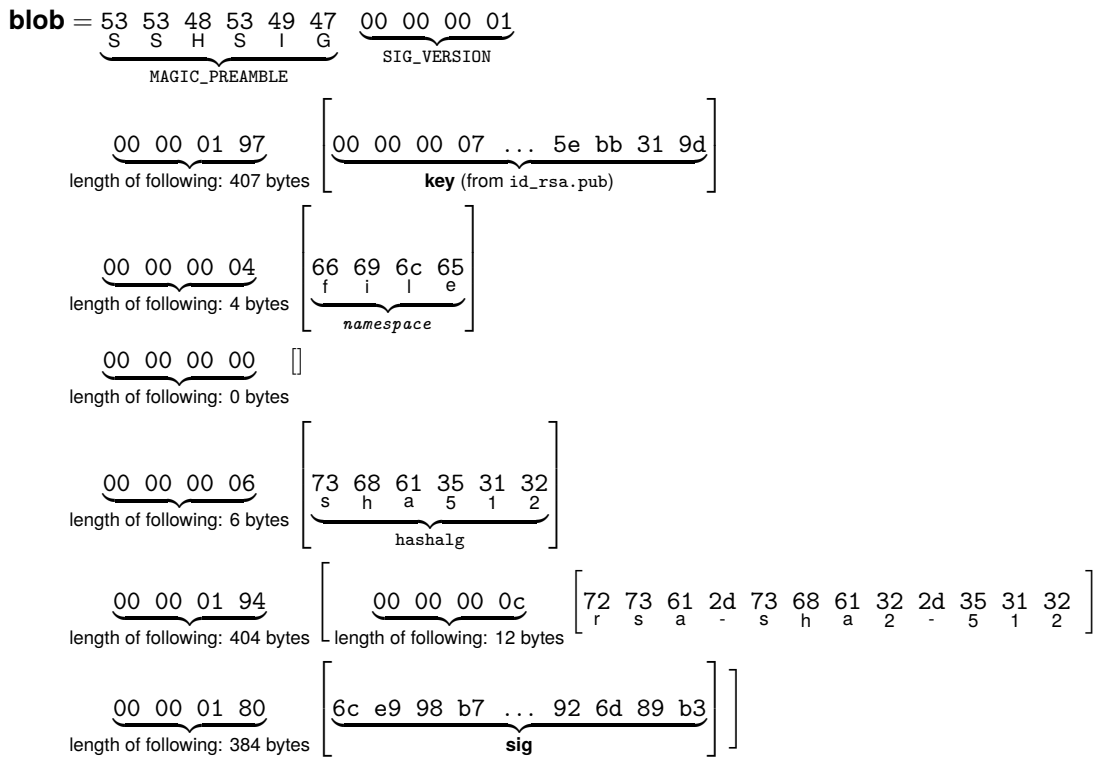
```
$ cat hello_world.txt
hello world
$ ssh-keygen -Y sign -f ~/.ssh/id_rsa -n file < hello_world.txt
Signing data on standard input
-----BEGIN SSH SIGNATURE-----
U1NIU0lHAAAAAQAAAZcAAAAHc3NoLXJzYQAAAAMBAAEAAAGBAORzFtOGKyFjxc0XCq+goE
LgUgobd54pwqqGo/V0p2V9AeWqai8Qy+lPCM7txDgvvxnPorBL/MSok+yV5+X4/SEVPrbI
JvJXaPgUHbdHqqRITEZcn5/fX8iysL98KcWrwD1wuKwntxJb0TA1sOgQ1t3sMIMXLxhIy7
9xoPWppjdHLZlrDmUfs+dcHM8GN4A7QA60jDLpkIaBGYbE73DU2uOFvugMWOELTnJ4ZeKL
dcTpbArTUf950sc23SzL5LDiViSYiCrHLpppDt0ZNm9Yl0g8OV/BrzoKb4UeHz0wnjd/X8
```

```
G6yfJYHSiTG3pGeC2dMI4kBOfEpbZoJlqjSXqi86LtsrLg85Ssz2XDJb6pxy7epOClbC9g
EMANSgPgrxr+K5+KPSkO2eZpFgkeQfSjr1YWQw+O8Yh71h2WPH7yE7l4LAZ5yQ5KF3XKE4
V6nK2Sux6HPtFidh5OSNlzEE++abqOoziK/u51roQFhyDJBwwSQCIe4pOD2XWg7HlPXrsx
nQAAAARmaWxlAAAAAAAAAAZzaGE1MTIAAAGUAAAADHJzYS1zaGEyLTUxMgAAAYBs6Zi3R2
sSpFkuzUYdev9JBPqx51La/szAm5NQ8Y9xpefj9drEbtqT4+1R6TcCSO3gW1EOCW2gYZeY
lTzTLoG3V/yfryRzJ2rxcGbNzVMbhnRVO5ZOkOqqNGnzbsJ5JzuJeDGN8v72HL1oXzN6BT
EFLg4iQDCtNmFiRxISEhGS35bL9Hti7+tvK4klUXI+MxPU++xjnt2ftCVSGYpQR+nOLBL+
GNImd2yLwEg9elrOBWRmHVv3uXnNL6Gqa7ExKwIC48XxZC+MQa0ApapqTh5mUfAXDAqveD
3iyBOZ9+aCCncNrzFU9y0OqUV4CWohfjMdmTB9YWL+zYBMGigwM9yXXJLNX6g5h3xDVqat
7rcCYJUiPBr7vZuG9+7qxPvAhMm72K33VIgpkgz4JXLWdRyaam3F6cVakFzbY/ve/JG84O
oLy2Eo415dBC6XEUp8OBxC2Laz54Y1jU34+YfRbvoqLm3NM3ffw3wTMpeXqHdo38UPAnW+
jhl2E5Uck5JtibM=
-----END SSH SIGNATURE-----
```

As before, the **blob** `U1NI...ibM=` is in Base64 format. Let us convert it to hexadecimal:

```
535348534947000000010000019700000007737368 2d727361000000030100010000018100e47316d3862b2163c5cd170aafa0a042e0520a1b779e29c2aa86a3f574a7657d01e5aa6a2f10cbe94f08ceedc4382fbf19cfa2b04bfcc4a893ec95e7e5f8fd21153eb6c826f25768f8141db747aaa4484c465c9f9fdf5fc8b2b0bf7c29c5abc03d70b8ac27b7125bd13035b0e810d6ddec3083172f1848cbbf71a0f5a9a637472d996b0e651fb3e75c1ccf0637803b400eb48c32e99086811986c4ef70d4dae385bee80c58e10b4e727865e28b75c4e96c0ad351ff79d2c736dd2ccbe4b0e2562498882ac72e9a690edd19366f5897483c395fc1af3a0a6f851e1f3d309e377f5fc1bac9f2581d28931b7a46782d9d308e2404e7c4a5b668265aa3497aa2f3a2edb2b2e0f394accf65c325bea9c72edea4e0a56c2f6010c00d4a03e0af1afe2b9f8a3d2934d9e66916091e41f4a3af5616430f8ef1887bd61d963c7ef213b9782c0679c90e4a1775ca13857a9cad92bb1e873ed162761e7448d973104fbe69bab4a3388afeee75ae84058720c9070c1240221ee29d03d975a0ec794f5ebb319d0000000466696c650000000000000006736861353132000001940000000c7273612d736861322d353132000001806ce998b7476b12a4592ecd461d7aff4904fab1e752dafeccc09b9350f18f71a5e7e3f5dac46eda93e3ed51e9370248ede05b510e096da0619798953cd32e81b757fc9faf2473276af17066cdcd531b8674553b964e90eaaa3469f36ec279273b8978318df2fef61cbd685f337a0531052e0e224030ad3661624712121211 92df96cbf47b62efeb6f2b892551723e3313d4fbec639edd9fb42552198a5047e9ce2c12fe18d226776c8bc0483d7a5af40564661d5bf7b979cd2fa1aa6bb1312b0202e3c5f1642f8c41ad00a5aa6a4e1e6651f0170c0aaf783de2c81d19f7e6820a770daf3154f72d0ea94578096a217e331d99307d6162fecd804c1a283033dc975c92cd5fa839877c4356a6adeeb7026095223c1afbbd9b86f7eeeac4fbc084c9bbd8adf7548829920cf82572d6751c9a6a6dc5e9c55a905cdb63fbdefc91bce0ea0bcb6128e35e5d042e97114a7c381c42d8b6b3e786358d4df8f987d16efa2a2e6dcd3377dfc37c13329797a87768dfc50f0275be8e197613951c93926d89b3
```

This breaks down as follows:



As we can see, **blob** consists of a bunch of padding prepended to **sig**. (It's worth noting that the public key is contained in **blob**, thus making the verification process entirely self-contained.)

Let us interpret **sig** as a nonnegative integer:

```
$ python3.12
Python 3.12.3 (main, Feb  4 2025, 14:48:35) [GCC 13.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> s=int("6ce998b7476b12a4592ecd461d7aff4904fab1e752dafeccc09b9350f18
f71a5e7e3f5dac46eda93e3ed51e9370248ede05b510e096da0619798953cd32e81b75
7fc9faf2473276af17066cdcd531b8674553b964e90eaaa3469f36ec279273b8978318
df2fef61cbd685f337a0531052e0e224030ad366162471212121192df96cbf47b62efe
b6f2b892551723e3313d4fbec639edd9fb42552198a5047e9ce2c12fe18d226776c8bc
0483d7a5af40564661d5bf7b979cd2fa1aa6bb1312b0202e3c5f1642f8c41ad00a5aa6
a4e1e6651f0170c0aaf783de2c81d19f7e6820a770daf3154f72d0ea94578096a217e3
31d99307d6162fecd804c1a283033dc975c92cd5fa839877c4356a6adeeb7026095223
c1afbbd9b86f7eeeac4fbc084c9bbd8adf7548829920cf82572d6751c9a6a6dc5e9c55
a905cdb63fbdefc91bce0ea0bcb6128e35e5d042e97114a7c381c42d8b6b3e786358d4
df8f987d16efa2a2e6dcd3377dfc37c13329797a87768dfc50f0275be8e197613951c9
3926d89b3",16)
>>> print(s)
247163529140781130332632461561139421433814262936044329483044058296138 5
318135901974984643537845947490685368344921396682912966109993414704528 1
352287323248777297873039324581399369139191795460242071746279749947503 3
808139699252657506283303977297214889191041121896714238089063999441291 9
223974047607628363470902524122180914964199587387624169606697708949308 1
866063775711966656995989171597573411928415481218905140576300011875092 3
964370076817457560331129771529718122039120224306947108129404679781242 4
378624858524213894140630870864466594276181431529167877809179352014158 9
341125465898535983176503868405984949829165172884171008937983645604734 5
673160104348869425697370987964186464681344944672690498829258438242455 2
366137016936778774774391993967297948509396493102764386169514872080353 0
```

```
4236638604339460756859804582864702404586746766335337210079081053324999
4878987915714802198787458145656239916145326268541775475404719425878197
696231480592819
```

Let us also interpret the modulus n as a nonnegative integer:

```
>>> n=int("e47316d3862b2163c5cd170aafa0a042e0520a1b779e29c2aa86a3f574a
7657d01e5aa6a2f10cbe94f08ceedc4382fbf19cfa2b04bfcc4a893ec95e7e5f8fd211
53eb6c826f25768f8141db747aaa4484c465c9f9fdf5fc8b2b0bf7c29c5abc03d70b8a
c27b7125bd13035b0e810d6ddec3083172f1848cbbf71a0f5a9a637472d996b0e651fb
3e75c1ccf0637803b400eb48c32e99086811986c4ef70d4dae385bee80c58e10b4e727
865e28b75c4e96c0ad351ff79d2c736dd2ccbe4b0e2562498882ac72e9a690edd19366
f5897483c395fc1af3a0a6f851e1f3d309e377f5fc1bac9f2581d28931b7a46782d9d3
08e2404e7c4a5b668265aa3497aa2f3a2edb2b2e0f394accf65c325bea9c72edea4e0a
56c2f6010c00d4a03e0af1afe2b9f8a3d2934d9e66916091e41f4a3af5616430f8ef18
87bd61d963c7ef213b9782c0679c90e4a1775ca13857a9cad92bb1e873ed162761e744
8d973104fbe69bab4a3388afeee75ae84058720c9070c1240221ee29d03d975a0ec794
f5ebb319d",16)
>>> print(n)
5184382712422635479859749601923596759264842227807945841481493073075146
5121894569796805673385263335759496789615627904148586275960577365759569
8440776655144672883502854592995598465061954177331945484865453696554540
8502109382330192301814641982367557293639738610001574650718140520475509
3721515504299754033536820040369667576320388248399820242006393579979341
8020249813640150579188554858941452947614274919327988002406274960929498
3354570768521178630194421967478070280582456598687708149700623943665420
4444758591977471233242951570080048212389807537314067184949511322511217
6066080504273862898622928564429220927561319229960834548969283869521355
6738125378106533556630892988428547541045518374183418355740271520485702
2288066791878553332041737958535376909380339384107721854507356484481140
3608455133067533830175434628836012840365379032079794616429368693719569
6214862850044827437544458145649293447074676167257668884251119739620861
830517055238557
```

Let us encrypt s using the public key (e, n) (recall that this is how we verify signatures):

```
>>> m=pow(s,65537,n) #Returns s^65537 (mod n)
>>> print(m)
1772951048391710834612885294587577627773124748645903087360072678310303
8729849579546317553846841792014316791252914369254894990629866150046147
6528493413910900910403797598463149989137263518952991552741844659638727
1848977563641364778084709650991632544644885622796339346280744392624431
7861876125522795346452366128824595701763259327106877380865051880638099
8792973335037698500048131945927015037188140569230699888127021967479443
5398865011487955338506642710127347139863196664927410461299562588600451
5762901144925108461929358856840939366915126831106858400678207231925708
8409266846109271757767637887715992340673266421158701936684749264473625
5937895167097973290879611699049818826354752924298100675066250839592142
9102056637534471407745067338448899785393153686637741585252793368275369
1563082709898633652760090211704445164499770634297117895486777123770351
8467188330700458214935954475033895356493655691483030317428503515192531
49129258568
>>> print(hex(m))
0x1fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

```
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffff003051300d060960864801650304020
305000440766f794844a18ccb789fa185268f645b88fe703b77ec9c75ace7ff24a78f1
d7a25e12544873b8aad18ebc033d5939ff8c7f488ed494d85b211733a871db63648
```

We see that $m \coloneqq s^e \pmod n$ has a very special form in hexadecimal. Here is the format:

$$m \text{ (length: 384 bytes)} \begin{cases} \underbrace{\texttt{00 01 ff ff ff ff ... ff ff ff ff}}_{\text{298 bytes, each of form ff}} \texttt{ 00 30 51 30} \\ \texttt{0d 06 09 60 86 48 01 65 03 04 02 03 05 00 04 40} \\ \underbrace{\texttt{76 6f 79 48 ... 1d b6 36 48}}_{h \text{ (64 bytes)}} \end{cases}$$

We see that m consists of 384 bytes, the same length as the public key n. Moreover, the first 320 bytes of m are set to a specific constant, whereas the final 64 bytes of m (denoted h) are a SHA-512 hash of something. Let us try to form a string whose hash is h.

Start with the original `msg` (recall that this was `hello world`). Let us compute its hash.

```
$ sha512sum hello_world.txt
db3974a97f2407b7cae1ae637c0030687a11913274d578492558e39c16c017de84eacd
c8c62fe34ee4e12b4b1428817f09b6a2760c3f8a664ceae94d2434a593  hello_worl
d.txt
```

So we have

$$\texttt{hash(msg)} = \underbrace{\texttt{db 39 74 a9 ... 24 34 a5 93}}_{\text{(64 bytes)}}.$$

Now let us form a specific string `tosign`:

$$\texttt{tosign} = \underbrace{\texttt{53 53 48 53 49 47}}_{\texttt{MAGIC\_PREAMBLE}} \quad \underbrace{\texttt{00 00 00 04}}_{\text{length of following: 4 bytes}} \left[\underbrace{\texttt{66 69 6c 65}}_{\textit{namespace}} \right] \quad \underbrace{\texttt{00 00 00 00}}_{\text{length of following: 0 bytes}} [] \quad \underbrace{\texttt{00 00 00 06}}_{\text{length of following: 6 bytes}} \left[\underbrace{\texttt{73 68 61 35 31 32}}_{\texttt{hashalg}} \right] \quad \underbrace{\texttt{00 00 00 40}}_{\text{length of following: 64 bytes}} \left[\underbrace{\texttt{db 39 74 a9 ... 24 34 a5 93}}_{\texttt{hash(msg)}} \right]$$

Note the similarities between `tosign` and **blob**.

What happens when we compute `hash(tosign)`?

$$\texttt{hash(tosign)} = \underbrace{\texttt{76 6f 79 48 ... 1d b6 36 48}}_{\text{64 bytes}} = h$$

We have successfully found something that hashes to h.

To recap, here is the entire SSH signature algorithm (using RSA-3072 and SHA-512), on inputs `msg` and *namespace*:

1. Calculate `hash(msg)` (length: 64 bytes)
2. Form `tosign`, using *namespace* and the previously calculated `hash(msg)`
3. Set $h \coloneqq$ `hash(tosign)` (length: 64 bytes)
4. Form m (length: 384 bytes), using h
5. Set $s \coloneqq m^d \pmod n$ (using the private key)
6. Let **sig** (length: 384 bytes) be s in hexadecimal
7. Form **blob**, using the public key, *namespace*, and **sig**
8. Convert **blob** to Base64, and enclose in "`---BEGIN SSH SIGNATURE---`" and "`---END SSH SIGNATURE---`"

And here is the algorithm for verifying a signature (assumed to be RSA-3072 and SHA-512), on inputs `msg`, *namespace*, and `file_containing_ssh_signature`:

1. Read from `file_containing_ssh_signature`, stripping away the "`---BEGIN SSH SIGNATURE---`" and "`---END SSH SIGNATURE---`" to obtain **blob**
2. Convert **blob** to hexadecimal
3. Obtain **key** from **blob**
4. Verify that *blob_namespace* matches *namespace*
5. Obtain **sig** from **blob**
6. Let s be **sig** interpreted as a nonnegative integer
7. Obtain n (the modulus) and e (the exponent) from **key**
8. Set $m \coloneqq s^e \pmod n$
9. Convert m to hexadecimal
10. Verify that m has length 384 bytes, and that the first $384-64 = 320$ bytes of m are `00 01 ff ff ... 05 00 04 40`
11. Let h be the final 64 bytes of m
12. Calculate `hash(msg)`
13. Form `tosign`, using *namespace* and the previously calculated `hash(msg)`
14. Verify $h =$ `hash(tosign)`

(For RSA-2048 and RSA-4096 keys, simply replace 384 by 256 and 512, respectively.)

5 Further reading

The man page for `ssh-keygen` is at https://man.openbsd.org/ssh-keygen.1

The source code for `ssh-keygen` is at https://github.com/openssh/openssh-portable/blob/master/ssh-keygen.c (scroll to the bottom to see the `main` method). Details of the signing procedure can be found in https://github.com/openssh/openssh-portable/blob/master/sshsig.c#L164 (the `sshsig_wrap_sign` method). Details of the verification procedure can be found in https://github.com/openssh/openssh-portable/blob/master/sshsig.c#L305 (the `sshsig_wrap_verify` method). The relevant code from `sshsig_wrap_sign` is copied below.

```
183         if ((r = sshbuf_put(tosign, MAGIC_PREAMBLE, MAGIC_PREAMBLE_LEN)) != 0 ||
184             (r = sshbuf_put_cstring(tosign, sig_namespace)) != 0 ||
185             (r = sshbuf_put_string(tosign, NULL, 0)) != 0 || /* reserved */
186             (r = sshbuf_put_cstring(tosign, hashalg)) != 0 ||
187             (r = sshbuf_put_stringb(tosign, h_message)) != 0) {
188                 error_fr(r, "assemble message to sign");
189                 goto done;
190         }
```

⋮

```
217         if ((r = sshbuf_put(blob, MAGIC_PREAMBLE, MAGIC_PREAMBLE_LEN)) != 0 ||
218             (r = sshbuf_put_u32(blob, SIG_VERSION)) != 0 ||
219             (r = sshkey_puts(key, blob)) != 0 ||
220             (r = sshbuf_put_cstring(blob, sig_namespace)) != 0 ||
221             (r = sshbuf_put_string(blob, NULL, 0)) != 0 || /* reserved */
222             (r = sshbuf_put_cstring(blob, hashalg)) != 0 ||
223             (r = sshbuf_put_string(blob, sig, slen)) != 0) {
224                 error_fr(r, "assemble signature object");
225                 goto done;
226         }
```

The document RFC 8332 https://datatracker.ietf.org/doc/html/rfc8332 gives a high-level overview of SSH signatures with RSA and SHA. See RFC 8017 Section 8.2 https://datatracker.ietf.org/doc/html/rfc8017#section-8.2 for details.