Rigorous specifications of the SSH Transport Layer

Erik Poll[1]⋆ and Aleksy Schubert[2]⋆⋆

[1] Digital Security, Institute of Computing and Information Science, Radboud University Nijmegen
[2] Institute of Informatics, Faculty of Mathematics, Informatics and Mechanics, Warsaw University

Abstract. This document presents (semi-)formal specifications of the security protocol SSH, more specifically the transport layer protocol, and describe a source code review of OpenSSH, the leading implementation of SSH, using these specifications.
Our specifications, in the form of finite state machines, are at a different level of abstraction that the typical formal descriptions used to study security protocols. Our motivation is to understand actual implementations of SSH, so we try to capture some of the details from the official (informal) specification that are irrelevant to the security of the abstract protocol, but which do complicate the implementation.
Our specifications should be useful to anyone trying to understand or implement SSH. First versions of our specifications were developed for the formal verification of a Java implementation of SSH [17].

1 Introduction

The SSH protocol is officially specified in a set of five RFCs, namely RFC 4250-4254 [16, 26–29]. Understanding the SSH protocol based on these RFCs is a daunting task, as they add up to about 150 pages. The SSH protocol is broken down into three layers – the transport, authentication, and connection protocols – which provides some modularity, but the layers are not quite independent.

This stands in shrill contrast with typical descriptions in the standard 'Alice-Bob' notation for security protocols. For example, the description of the SSH Transport Layer Protocol in [25] is only six lines. Of course, such a description ignores some details and abstracts away from others. This is the whole point of such descriptions: these concise abstract presentations are meant to be useful for understanding and analysing security properties of the protocol, possibly using one of the many tool-supported formal methods that are available to analyse security protocols (e.g., [1, 2, 6, 8, 9, 21, 24]).

Still, when faced with the job of implementing the protocol, or understanding an existing implementation of SSH, e.g. as part of security review, one cannot ignore or abstract away from irrelevant details in the RFCs. In this report we explore the possibilities for formal specifications of SSH, which capture more details of the RFCs than the standard "Alice-Bob" notation, without resorting to a dozens of pages of English prose.

Some of the tricky issues we want to capture include the handling of messages that are ill-formed, messages that arrive out of sequence, and messages for optional parts of the protocol. We also want to take into account the asynchronous nature of communication. All these details are typically ignored when analysing security of the abstract protocol, but an implementation still has to get them right.

An important complication is that SSH is really a family of protocols: there are some optional parts, some restrictions on which combinations of optional parts are allowed, and it is parameterised by other protocols. For instance, several key exchange protocols can be used.

Another complication is that the official specifications are not always clear in what the response to an unexpected, unsupported message should be: some of these *may* or *should* be ignored, whereas others *must* lead to disconnection. Information about this is spread over the various RFCs and in many cases implicit, complicating the job of anyone implementing the standard.

⋆ Supported by the Sixth Framework Programme of the EU under the MOBIUS project FP6-015905.
⋆⋆ Supported supported by the Sixth Framework Programme of the EU under the SOJOURN project MEIF-CT-2005-024306.

Underspecification in the specifications can be dealt with in various ways. Some people advocate the *robustness principle*, also know as Postel's Law: "be conservative in what you send, liberal in what you accept" [18, Section 3.2]. However, given that the security provided by security protocols can be very fragile, it seems better to adopt the *correctness principle*: "be conservative in what you send, *and* conservative in what you accept". Note that there has been quite some debate about the precise interpretation of Postel's law and the way it is used as a poor excuse for imprecise specs or non-conformant implementations.

Our goal is to provide a (semi)formal definition of SSH Transport Layer Protocol that captures the issues mentioned above, so that it could either act as a detailed blueprint for an implementation, or – as we have used it – as a basis for performing a thorough code review of an implementation. To describe the protocol we use finite state machines. The notion of 'state' is crucial for the correctness of the protocol, but largely implicit in the official specifications. Making it explicit is an important step towards understanding of possible implementations. Ideally, such finite state machine descriptions in specification could provide the starting point for implementations [14].

We started this work during an exercise in formal program verification, in which we verified a Java implementation of SSH, where we were confronted with the issues described above. This verification effort, using the program verification tool ESC/Java2 and the formal specification language JML is reported in [17]. It revealed this Java implementation to be completely insecure; it did not keep track of any protocol-state whatsoever. The models described in this paper have been used for an informal code review of OpenSSH, which is described in Section 9.

The rest of the paper is organised as follows: Section 2 presents a general overview of the SSH protocol and the documents which standardise it. Section 3 presents a general framework of the SSH Transport Layer Protocol in the form of "Alice-Bob" interaction. Then we provide stepwise refinements for the protocol description in terms of finite state machines. Section 4 presents an initial model, which is then refined

- in Section 5 by considering the parallelism and asynchronous communication between client and server,
- in Section 6 by including the possibility to guess the key exchange algorithm,
- in Section 7 by allowing for key re-exchange, and
- in Section 8 with the different categories of unexpected messages.

Section 9 then describes the code review of OpenSSH. Finally, sections 10 and 11 discuss related work and conclude.

2 Overview of SSH

SSH is defined in five RFCs: RFCs 4250-4524 [16, 26–29]. We will use the symbolic names to refer to these RFCs – [SSH-NUMBERS], [SSH-ARCH], [SSH-TRANS], SSH-USERAUTH], and [SSH-CONNECT] – as is also done in the RFCs.

Two first two RFCs describe common notations and the overall architecture:

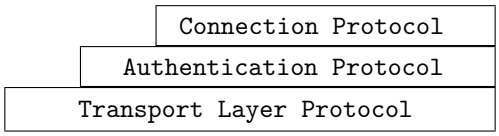
- [SSH-NUMBERS], the **SSH Protocol Assigned Numbers**, summarises the numbers and symbolic names used in the protocol, e.g. for message numbers, error messages, etc.
- [SSH-ARCH, the **SSH Protocol Architecture** describes the architecture of the protocol, fixes the terminology and discusses the security objectives of SSH.

The other three RFCs then describe the three sub-protocols that make up the layers of the SSH protocol stack:

- [SSH-TRANS] defines the **SSH Transport Layer Protocol**, the sub-protocol for establishing a connection. This sub-protocol negotiates an algorithm, establishes session keys, authenticates the server and finishes with the initialisation of the SSH data exchange. It ensures authentication of the server and confidentiality & integrity of the communication.

– [SSH-USERAUTH] defines the **SSH Authentication Protocol**, the sub-protocol to establish the authenticity of the user who is about to log in with the use of SSH, e.g. by username/password.
– [SSH-CONNECT] defines the **SSH Connection Protocol**, the sub-protocol to establish different communication channels within an SSH session (e.g. port forwarding, terminal, X11 communication) together with parameters of the channels.

The three sub-protocols are run in the order as they are listed above: first the Transport Layer Protocol is used to establish a connection, then the Authentication Protocol is started to authenticate the user, and finally the Connection Protocol is used to establish sessions of different services that SSH provides. The three sub-protocols are not simply run consecutively, but rather 'on top' of each other: the Authentication Protocol on top of the Transport Layer Protocol, and the Connection Protocol on top of the authentication protocol:



With the exception of the very first messages of the Transport Layer Protocol, all protocols use the same format for packets, the so-called **Binary Packet Protocol** defined in [SSH-TRANS, §6]. Here one byte in each packet is the message number, which determines the type of message. Different ranges of message numbers are then reserved for the various sub-protocols: 1-49 for the Transport Layer Protocol, 50-79 for User Authentication Protocol, 80-127 for the Connection Protocol, with 128-255 reserved for client protocols and local extensions.

Messages of the different sub-protocols may or may not be allowed at various stages. Messages that are specific to the Transport Layer Protocol, notably `SSH_MSG_KEXINIT` to restart a key exchange, can occur at any time during other stages.

2.1 The Transport Layer Protocol

The Transport Layer Protocol guarantees the central security objectives of SSH, namely confidentiality & integrity of the communication. It can be further divided into 4 stages:

1. **the protocol identification phase**, to decide which version of SSH – SSH1 or SSH2 – is run (steps 1-3 in Fig. 1);
2. **the algorithm negotiation phase**, to decide which algorithm is used for key exchange (steps 4 and 5 in Fig. 1);
3. **the key exchange phase**, to do the actual key exchange using this algorithm (steps 6 and 7 in Fig. 1);
4. **the service request phase**, which starts the subsequent protocols (anything after step 9 in Fig. 1).

The Transport Layer Protocol is still parametrised by an algorithm for key exchange. [SSH-TRANS] prescribes two obligatory key exchange algorithms, called diffie-hellman-group1-sha1 and diffie-hellman-group14-sha1, which differ in the group used for the Diffie-Hellman key exchange computations. RFC 4419 [11] adds the possibility to negotiate a more secure group for the Diffie-Hellman key exchange. RFC 4462 [15] specifies Diffie-Hellman key exchange using the Generic Security Service Application Program Interface (GSS-API). RFC 4432 [13] and RFC 5656 [22] describe key-exchange algorithms based on RSA and Elliptic Curve Cryptography (ECC), respectively.

3 The SSH Transport Layer Protocol in "Alice-Bob" style (Version 1)

Figure 1 describes the SSH2 Transport Layer Protocol using Diffie-Hellman key exchange in the common "Alice-Bob" notation for security protocols. Rather than using A for Alice and B for Bob,

1. $C \to S$: `CONNECT`
2. $S \to C$: `VERSION_S` server version string
3. $C \to S$: `VERSION_C` client version string

} protocol identification

4. $S \to C$: `SSH_MSG_KEXINIT` I_C
5. $C \to S$: `SSH_MSG_KEXINIT` I_S

} key exchange algorithm negotiation

6. $C \to S$: `SSH_MSG_KEXDH_INIT`e
   where $e = g^x$ for some client nonce x
7. $S \to C$: `SSH_MSG_KEXDH_REPLY`$K_S, f, sign_{K_S}(H)$
   where $f = g^y$ for some server nonce y,
   $K = e^y$ and $H = hash(V_C, V_S, I_C, I_S, K_S, e, f, K)$,
   K_S is the server key
8. $S \to C$: `SSH_MSG_NEWKEYS`
9. $C \to S$: `SSH_MSG_NEWKEYS`

} key exchange

10. ...

} session, incl. SSH authentication and connection protocols

Fig. 1. The abstract Transport Layer Protocol of SSH2 with Diffie-Hellman key exchange.

we use C for Client and S for Server. The description assumes both parties support SSH version 2 and want to use Diffie-Hellman as the key exchange algorithm.

The RFCs do not always clearly distinguish the negotiation of key exchange algorithm negotiation (steps 4 and 5) from the subsequent key exchange (steps 6-9), and sometimes use the term 'key exchange' to refer to both phases together.

All messages, or packets, after the protocol identification phase are in format prescribed by the **Binary Packet Protocol** [SSH-TRANS, §6], where one byte in each packet is the message number, which determines the type of message. Fig. 1 abstracts from this format, but does include the symbolic names of the message numbers, such as `SSH_MSG_KEXINIT`, as defined in [SSH-NUMBERS]. These symbolic names all start with `SSH_MSG`; we will omit this prefix from now on to avoid clutter.

After the protocol in Fig. 1 is completed, subsequent traffic between client and server is encrypted and digitally signed. The four keys for this — for each party one for signing and one for encryption — are derived from K and H. The encryption and signing algorithms used are determined on the basis of I_C and I_S.

After the protocol in Fig. 1 is completed, at any stage either party can send a `SSH_MSG_KEXINIT` message to re-negotiate a new session key.

The description in Fig. 1 abstracts from some aspects and ignores others, such as the actual format of messages, the encryption of the session after step 9, and the possibility of key re-exchange, but also the asynchronous nature of communication and alternative protocol runs that are allowed by the specification. In the following sections try to capture these complications. We will continue to abstract from the actual format of messages. Indeed, we will go one step further and abstract from their content altogether, except for the symbolic names of the message numbers. So we focus on the possible sequences of messages that are correct, but ignore the actual contents of these messages.

The issues considered in the following sections include:

- the parallelism between client and server, and the asynchronous nature of the communication between the two parties, which for instance allows both parties to send their version strings simultaneously;
- the possibility of guessing the key exchange algorithm, and optimistically sending the first key exchange packet before negotiation of the key exchange algorithm has been completed, as allowed by the specs;
- the possibility of refreshing the session keys;
- dealing with 'unexpected' – incl. unsupported – messages.

4 Parallelism (Version 2)

The "Alice-Bob" notation in Fig. 1 over-specifies, in that it prescribes an order between messages that could be sent in a different order or in parallel. E.g., it specifies an order between the two `VERSION` messages, the two `KEXINIT` messages, and the two `NEWKEYS` operations, whereas the RFCs leave it open in which order these messages occur. This can be described as follows

```
CONNECT ;
(VERSION_C || VERSION_S) ;
(KEXINIT_C || KEXINIT_S) ;
KEXDH_INIT ;
KEXDH_REPLY ;
(NEWKEYS_C || NEWKEYS_S) ;
...
```

where `||` denotes parallel composition and `;` sequential composition. For messages that both parties can send, such as `KEXINIT`, we use suffixes `_C` and `_S` to indicate the party that sent them.

Instead of a textual representation, as given above, we can also use a state diagram to describe this, as is done in Figure 2. Advantages of this graphical notation are that it is easy to name states and to include cycles. To illustrate this, Fig. 2 also describes the possibility of key exchange (where the key is renegotiated after parties exchange `KEXINIT` messages) and the ongoing session as the messages `traffic_C` and `traffic_S` in state 6 (abstracting from the actual content of these messages)[3].

In an actual implementation of the protocol, the state will have to be recorded by the program point and/or by values of program variables. For example, in OpenSSH, the state is (sometimes) characterised by an array of 256 function pointers, as discussed in more detail in Section 9. Understanding how this implementation of the state relates to the abstract states in Fig. 2 is crucial to understanding the correctness of the implementation.

5 Asynchronisity (Version 3)

The asynchronous nature of communication between client and server gives us some freedom in resolving the possible parallelism. For example, both parties could send their `VERSION` messages simultaneously, so that both process the incoming message `VERSION` message of the other party after sending their own.

One natural way to do this is to *send all outgoing messages that can be sent before handling any incoming traffic*. This approach, which we will call *'priority to sending'*, is natural because it is efficient: waiting for incoming messages may waste time, or worse, result in a deadlock if both parties decide to wait for the other. Also, incoming traffic will typically be buffered, so there is no harm in postponing the handling of incoming traffic. Resolving the parallelism in this way gives the following descriptions for client and server:

```
client                              server

  CONNECT! ;                          CONNECT? ;
  VERSION_S? ; VERSION_C! ;           VERSION_S! ; VERSION_C? ;
  KEXINIT_C! ; KEXINIT_S? ;           KEXINIT_S! ; KEXINIT_C? ;
  KEXDH_INIT! ;                       KEXDH_INIT ? ;
  KEXDH_REPLY? ;                      KEXDH_REPLY! ;
  NEWKEYS_C! ; NEWKEYS_S? ;           NEWKEYS_S! ; NEWKEYS_C? ;
```

[3] Note that some arrows in Fig. 2 are labelled with the parallel composition of protocol steps. We could draw these out as individual steps, resulting in a diamond shape with one path for each possible interleavings. But this makes the diagram needlessly complex, and less accurate, as there can be true concurrency between these events.

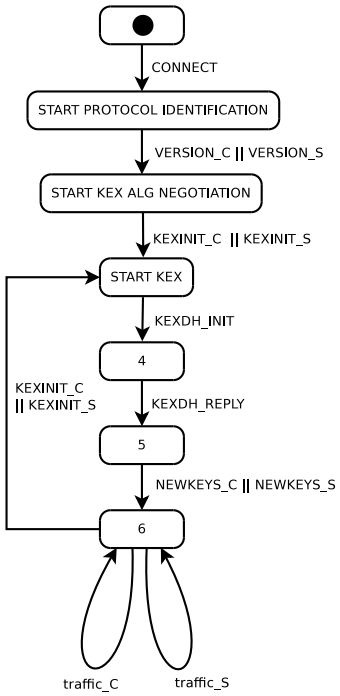


Fig. 2. Abstract description of SSH2 with Diffie-Hellman key exchange.

The suffix `!` means a message is sent, the suffix `?` that it is received.

Note that the order of events is different when seen from the point of view of the client or server. For example, the client sends `VERSION_C` before receiving `VERSION_S`, but the server sends `VERSION_S` before receiving `VERSION_C`. Fig. 3 gives a graphical representation of the textual description above. For the moment we ignore key re-exchange.

Many variations are possible in the graphical representation. For example, one could replace the arrow labelled `VERSION_C!;VERSION_S?` by two arrows, one labelled `VERSION_C!` and one labelled `VERSION_S?`, and introduce an extra state in between. Our choice not to do this is completely arbitrary. Our main motivation here is to keep successive refinements of the diagram similar and as readable as possible.

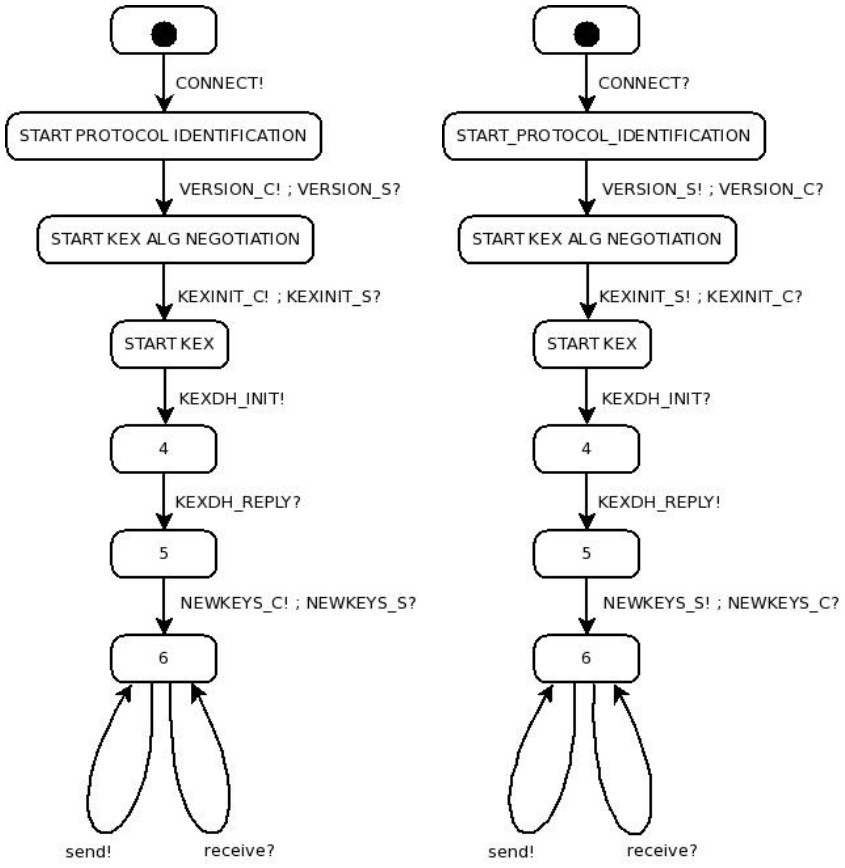
OpenSSH does not always take the "priority to sending" approach. For instance, the OpenSSH client waits for `VERSION_S` before sending `VERSION_C`.

6 Guessing the key exchange algorithm (Version 4)

The RFCs are a bit more liberal than the description above. Directly after sending its `KEXINIT` messages, each party may already guess the key exchange protocol, without waiting for the other party's `KEXINIT` to complete key exchange algorithm negotiation, and optimistically send a first message of the key exchange protocol, assuming of course that it is appropriate for that party to send the first message in the key exchange[4].

For a client wanting to do Diffie-Hellman key exchange this means it can send `KEXDH_INIT` directly after sending its `KEXINIT_C`, without waiting to receive `KEXINIT_S`. Since diffie-hellman-group1-sha1 and diffie-hellman-group14-sha1 have to be supported by all servers an optimistic guess has a good chance to be correct.

[4] Liberal reading of [SSH-TRANS] could even be interpreted as meaning that the first key exchange message could be sent *before* `KEXINIT`, but that is clearly silly.



(a) client (b) server

Fig. 3. SSH2 using Diffie-Hellman key exchange, resolving parallelism by giving priority to sending.

However, the client should also be prepared to receive a first key exchange packet from a server that makes a (wrong) guess for the key exchange protocol. Combining these aspects yields the description in Fig. 4. Here square brackets denote an optional message, which may occur at most once, and `KEX_WRONG_GUESS_S` is an erroneous first key exchange packet sent by the server. (The parallelism has not been resolved here; we will do that in the next section, when we also include key re-exchange.)

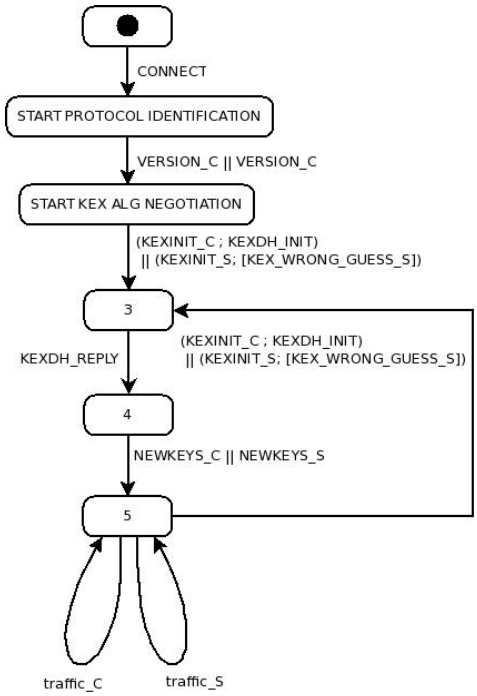


Fig. 4. SSH2 doing Diffie-Hellman key exchange, allowing for guessing of the key exchange algorithm (i.e. `KEXDH_INIT` may be sent before `KEXINIT_S`) and an erroneous initial key exchange packet by the other party (the optional `KEX_WRONG_GUESS_S` that is ignored).

7 Key re-exchange (Version 5)

Fig. 7 describes the full behaviour of a client and server, taking into account key re-exchange. Parallelism is resolved not taking the 'priority-to-sending' approach, but as is done in OpenSSH: the OpenSHH client waits for `VERSION_S` before sending `VERSION_C` and waits for `KEXINIT_S` before sending `KEXDH_INIT`.

Note that dealing with re-exchange of the keys in Fig. 7 is more involved than suggested in Fig. 4. This is because the official specs stipulate that

> `Note, however, that during a key re-exchange, after sending a SSH_MSG_KEXINIT message, each party MUST be prepared to process an arbitrary number of messages that may be in-flight before receiving a SSH_MSG_KEXINIT message from the other party. [SSH-TRANS, Sect 7.1, page 20]`

So in state `CLIENT_REKEX`, after the client has taken the initiative to refresh the keys, it should be prepared to receive incoming traffic form the server until it receives `KEXINIT_C`.

Dually, in state `SERVER_REKEX`, after the client receives `KEXINIT_S` from the server requesting the keys to be refreshed, there is nothing to prevent the client from using the old keys for a while

to send traffic. Whether a client should do this is of course questionable; hence the use of a dashed arrow in the diagram for this. It would be better to immediately respond with `KEXINIT_C!` and proceed with the key re-exchange. Of course, some messages sent by the client using the old keys may still be underway to server at this point.

8 Unexpected messages

What is still implicit in all the specifications up to now is what should happen if we receive a message different from the one we expect. For protocol specifications in "Alice-Bob" notation, as the one in Fig. 1, the implicit idea is that the protocol should be aborted if anything else happens. However, in reality the situation is more complicated Three responses are possible – or may be required – in case of an unexpected message:

1. the session is aborted, either by simply stopping all further communication, or, more gracefully, by sending a last `SSH_MSG_DISCONNECT` message before stopping all further communication;
2. the message is ignored, with the session continuing as if the message was never received;
3. the message is ignored, but the other party is informed about this, by replying with an `SSH_MSG_UNIMPLEMENTED` message, with the session then continuing as if these messages never happened [SSH-TRANS, §11.4].

Aborting the connection when an unexpected message is received is the safest thing to do. However, simply *ignoring* unexpected messages does not compromise the overall security, as long as expected messages with the "wrong" content (e.g. with incorrect signatures) *do* lead to disconnection. The real danger in a faulty implementation lies in processing an unexpected message as if were part of a legitimate protocol run.

In general, the RFCs leave quite some implementation freedom here. One approach is to follow the so-called *robustness principle*, also known as Postel's Law[5]

> Be liberal in what you accept, and conservative in what you send.

However, the *robustness principle* has come in for quite some criticism over the years, as a cause of – or excuse for – compliance problems in the long run. Being too liberal can be dangerous when performing security-sensitive operations, so here it may be better to:

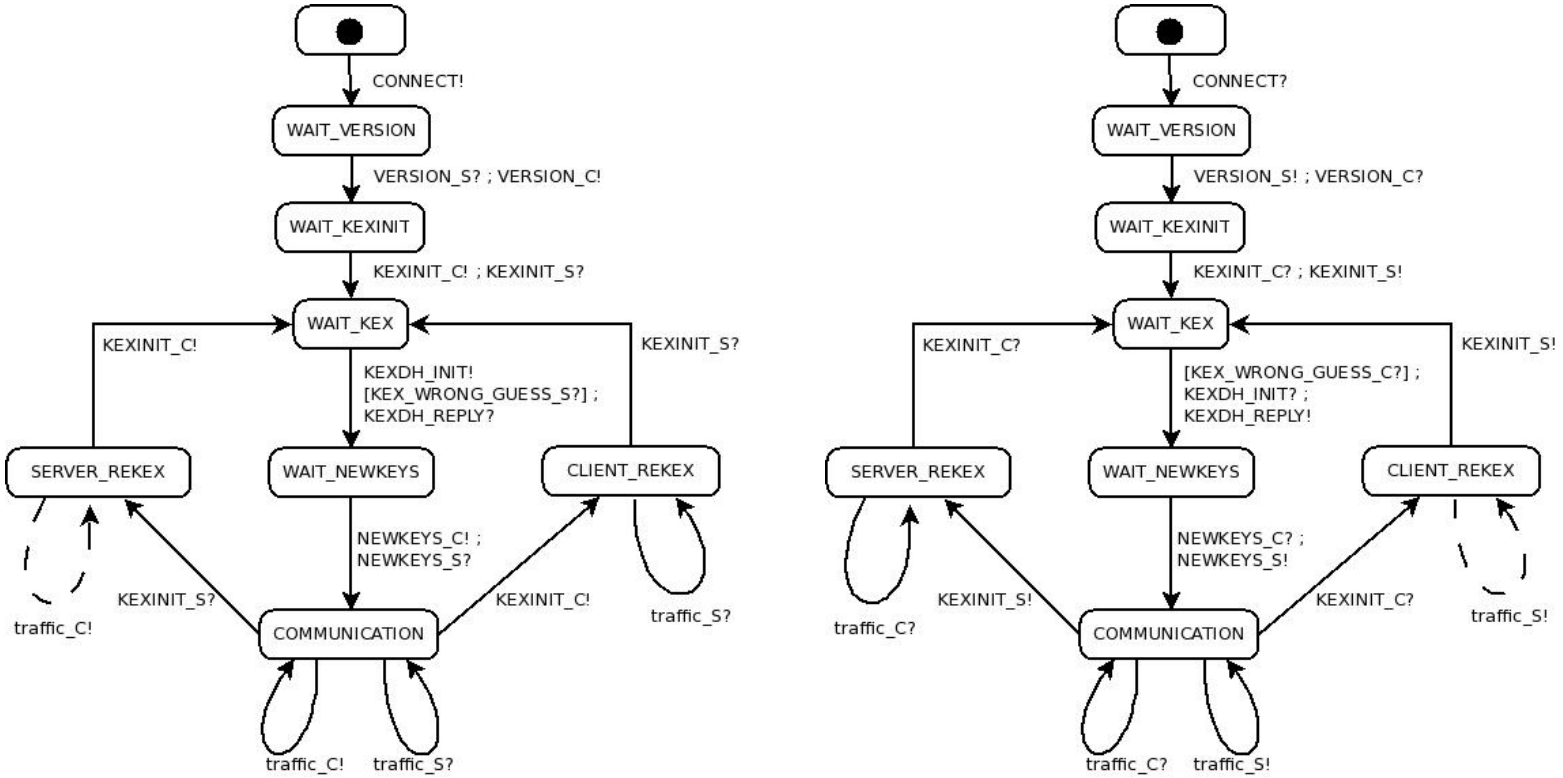
> Be conservative in what you accept, and conservative in what you send.

An inverse to Postel's Law, "Be conservative in what you accept, and liberal in what you do" has been proposed as a strategy to expose security flaws [10].

Below we consider the various types of unexpected messages – i.e. deviations from the normal protocol runs as given in the specifications so far – that there are and discuss how these should be dealt with:

1. *Expected messages with wrong content*
   Clearly, if we get a message of the right type (i.e. a message with the right message number), but with the wrong contents, then we must abort. For instance, if in state 3 in Fig. 4 the client receives a `KEXDH_REPLY` message with the wrong content – i.e. a wrong signature $sign_{K_S}(H)$ as defined in Fig. 1 – it should abort.
2. *Request for deconnection:* `SSH_MSG_DISCONNECT`
   At any stage we should be ready to accept a `SSH_MSG_DISCONNECT` message, and this should lead to immediate termination of the connection [SSH-TRANS, §11.1].
3. *Ignorable messages:* `SSH_MSG_IGNORE` `SSH_MSG_UNIMPLEMENTED`, `SSH_MSG_DEBUG`
   There are three types of messages that can always be ignored:

[5] The origin of this principle is RFC 793, Section 2.10: "TCP implementations will follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others".



(a) client (b) server

Fig. 5. SSH2 client and server supporting Diffie-Hellman key exchange, allowing for an erroneous initial key exchange packet by the other party. In resolving the possible parallelism the same choices are taken as in OpenSSH: (i) the client waits for the server to send its `VERSION` message before sending its own `VERSION`; (ii) the client does not use the possibility of guessing the key exchange algorithm, and hence sends its `KEXDH_INIT` only after receiving `KEX_INIT` from the server; (iii) the server waits for the client's `KEX_INIT` before replying with its own `KEX_INIT`.

- SSH_MSG_IGNORE and SSH_MSG_UNIMPLEMENTED MUST be ignored [SSH-TRANS, §11.2 and 11.4]
- SSH_MSG_DEBUG MAY be ignored [SSH-TRANS, §11.3].

An implementation could simply filter out these three types of messages in the incoming traffic. In fact, this is what the OpenSSH implementation does.

4. *The wrongly guessed KEX_WRONG_GUESS*
   As explained earlier, parties should be prepared to accept and ignore an erroneous key exchange message sent after an optimistic (but wrong) guess of the key exchange algorithm by the other party. We indicated this as an optional KEX_WRONG_GUESS transition in our specifications. However, KEX_WRONG_GUESS is not a fixed message number, which leaves the question: which messages from should be treated as an erroneous key exchange message?
   The most liberal approach is two treat any message as erroneous key exchange message and ignore it, except the two messages that are expected – SSH_MSG_KEX_INIT and SSH_MSG_KEXDH_REPLY – and SSH_MSG_DISCONNECT, of course.
   One could be more restrictive, and only treat messages with numbers in the rage 1-49 (the numbers reserved for the transport layer protocol) except SSH_MSG_KEX_INIT and SSH_MSG_KEXDH_REPLY as erroneous key exchange messages, and abort the session for all messages outside this range.
   Or, more restrictive still, one could restrict this to messages in the range 30-49, which are the numbers reserved for messages for specific key exchange methods.
5. *Other unexpected messages*
   This leaves the question of how to deal with all other unexpected messages that might be received at some point. We could choose to always abort the connection, or always ignore them. However, [SSH-TRANS, Sect 11.3] states:

   > An implementation MUST respond to all unrecognised messages with an SSH_MSG_UNIMPLEMENTED message in the order in which the messages were received. Such messages MUST be otherwise ignored. Later protocol versions may define other meanings for these message types.

   The big problem with interpreting this is that it is not clear at all which messages should be regarded treated as 'unrecognised messages'. Here we have a similar range of choices as in 4 above: should we consider all numbers currently not allocated in the specification as unrecognised messages, or only those in the range 1-49?

Some remarks in the RFCs suggest some further restrictions in the choices we can make choices for 4) and 5): [SSH-TRANS, §7.1] states

> Once a party has sent a SSH_MSG_KEXINIT message for key exchange or re-exchange, until it has sent a SSH_MSG_NEWKEYS message (Section 7.3), it MUST NOT send any messages other than:
> - Transport layer generic messages (1 to 19) (but SSH_MSG_SERVICE_REQUEST and SSH_MSG_SERVICE_ACCEPT MUST NOT be sent);
> - Algorithm negotiation messages (20 to 29) (but further SSH_MSG_KEXINIT messages MUST NOT be sent);
> - Specific key exchange method messages (30 to 49).
>
> The provisions of Section 11 apply to unrecognised messages.

and [SSH-AUTH, §6] states

> Message numbers of 80 and higher are reserved for protocols running after this authentication protocol, so receiving one of them before authentication is complete is an error, to which the server MUST respond by disconnecting, preferably with a proper disconnect message sent to ease troubleshooting.

The first comment justifies aborting the protocol when receiving any messages outside the range 1-49 while performing key exchange, i.e. *after* receiving SSH_MSG_KEXINIT from the other party *until* the SSH_MSG_NEWKEYS messages are exchanged. Although the second comment is given in the specification of the authentication protocol – i.e. [SSH-AUTH], and not [SSH-TRANS] –, as the transport layer protocol is also run 'before authentication is complete', it could be taken to apply to the transport layer protocol too.

9 Code review of OpenSSH

We did a manual code review to check that OpenSSH does indeed implement the protocols as described in Figure 7, for both client and server. In the course of doing this we recorded a detailed description of how the OpenSSH implementation works, which takes up the remainder of this section. We recorded this information for our own benefit – without it we really kept getting lost in the code. For people who not interested in the working of OpenSSH skimming this section might provide some idea of the complexities involved in such a code review.

To understand the code we combined a top-down approach (following the control flow down from the `main` procedures) with a bottom-up approach (looking for the procedure responsible for handling incoming or outgoing messages and then proceeding up the call chain to where these are used), documenting the procedures of interest along the way.

The OpenSSH source code consists of well over a hundred files. In the end, the functionality we were interested was involved around a dozen of these files, namely

- `ssh.c`
- `sshd.c`
- `sshconnect.c`
- `sshconnect2.c`
- `dispatch.c`
- `packet.c`
- `serverloop.c`
- `clientloop.c`
- `kexdhc.c`
- `kexgex.c`

A major complicating factor in tracing the control flow was the use of function pointers, in the so-called global dispatch table, as explained below.

Conventions Procedure names are written with trailing `()`, e.g. `packet_read_inspect()`, to distinguish them from file or variable names. Arguments and argument types are omitted.

Procedure names in the OpenSSH source code often start with the name of the file in which that procedure is defined, e.g. `packet_read_inspect()` is in the `packet.c`. We only mention which file a procedure is when this convention is not followed.

Message names for SSH2 start with `SSH2_MSG`; we usually omit this, writing `KEX_INIT` instead of `SSH2_MSG_KEX_INIT`. A subscript C or S is sometimes added to a message name to make it explicit if it is sent by the Client of the Server.

Handling incoming messages There are two procedures in OpenSSH to handle incoming traffic, namely `dispatch_run()` or `packet_read_expect()`. The latter is used when an incoming message of a specific type is expected, the former when messages of different types may arrive:

- `dispatch_run()`:
  This procedure uses a dispatch table, a global array `display` of function pointers, one for every message type. It retrieves an incoming message, reads the byte that specifies the message type, and then forwards handling of the message to the corresponding entry in the dispatch table.
  The content of the dispatch table is changed at various stages during the protocol, described in more detailed later Three helper procedures are used to do this: `dispatch_range()`, `dispatch_set()`, and `dispatch_init()`.
  For retrieving the incoming message, `dispatch_run()` calls down to `packet_read_poll_seqnr()` or `packet_read_seqnr()`. Both these methods take care of handling the generic messages `SSH2_MSG_IGNORE`, `SSH2_MSG_DEBUG`, `SSH2_MSG_DISCONNECT`, and `SSH2_MSG_UNIMPLEMENTED` in the appropriate way.

– `packet_read_expect(int expected_type)`:
  This procedure aborts the connection if a message is received of a type other than the specified type.
  It calls down to `packet_read_seqnr()`, which – as already mentioned above – takes care of the generic messages types `SSH2_MSG_IGNORE`, `SSH2_MSG_DEBUG`, `SSH2_MSG_DISCONNECT`, and `SSH2 _MSG_UNIMPLEMENTED` in the appropriate way.

Clearly these approaches are very different when it comes to handling unexpected messages: `packet_read_expect()` will disconnect, whereas what `dispatch_run()` does depends on the current content of the dispatch table. Often large parts of the dispatch table are filled with pointers to error handling procedures, e.g. `dispatch_protocol_error()` or `kex_protocol_error()`.

9.1 The client

The `main()` program for the client in `ssh.c` calls `ssh_login()` (in `sshconnect.c`) to do the key negotiation and the user authentication. It then calls `ssh_session2()`, which in turn calls `client_loop()` in `clientloop.c` for the interactive session.

To carry out the transport layer protocol, the procedure `ssh_login()`

– calls `ssh_exchange_identification()` in `sshconnect.c` to exchange version numbers: it waits for the server's identification string and then sends the client's; VERSION$_S$?
– calls `ssh_kex2` in `ssh2connect.c` for the key exchange, discussed below; VERSION$_C$!
– finally calls `ssh_userauth2()` in `ssh2connect.c` to handle the session.

For the key exchange, the procedure `ssh_kex2`

– defines a `Kex`-struct `kex` with function pointers to the procedures that do the key exchange, which are `kexdh_client()` and `kexgex_client()`;
– calls `kex_setup()`, which in turn
  • calls `kex_send_kexinit()` to send `SSH2_MSG_KEXINIT`; KEXINIT$_S$!
  • calls `kex_reset_dispatch()` to reset the dispatch table, setting all transport protocol messages (the range 1- 49) except `KEX_INIT` to be treated as errors;
– calls `dispatch_run()`.

Now `dispatch_run()` will only respond to `KEXINIT`, and hand over control to the procedure `kex_input_kexinit()` to handle it. This procedure KEXINIT$_S$? KEXINIT$_C$?

– sends a `SSH2_MSG_KEXINIT`, if the client hasn't done so already; this cannot be the case when `kex_input_kexinit()` is called the first time to set up a new session, but it can be the case when `kex_input_kexinit()` is invoked later to refresh the keys; KEXINIT$_S$!
– calls a procedure from the `kex` struct `kexdh_client()` and `kexgex_client()`, for the actual key exchange,

The procedure `kexdh_client()` then

– sends `SSH2_MSG_KEXDH_INIT`; KEXDH_INIT!
– receives `SH2_MSG_KEXDH_REPLY` using `packet_read_expect`; KEXDH_REPLY?
– calls `kex_finish()` first sends `SSH2_MSG_NEWKEYS` and then receives `SSH2_MSG_NEWKEYS` using `packet_read_expect()`. NEWKEYS$_C$! NEWKEYS$_S$?

Alternatively, the procedure `kexgex_client()`

– sends `SSH2_MSG_KEX_DH_GEX_INIT`; GEX_INIT!
– receives `SH2_MSG_KEX_DH_GEX_REPLY` using `packet_read_expect()`; GEX_REPLY?
– calls `kex_finish()` which sends `SSH2_MSG_NEWKEYS` and then receives `SSH2_MSG_NEWKEYS` using `packet_read_expect()`. NEWKEYS$_C$! NEWKEYS$_S$?

So, as far as the incoming messages for transport layer protocol are concerned, the client handles KEXDH_REPLY and NEWKEYS using packet_read_expect(), and only uses dispatch_run() to handle incoming KEX_INIT messages. This makes sense, as only KEX_INIT can be received when other messages (namely messages of higher protocol layers) are expected.

From the moment that re-keying starts (by sending KEXINIT_C) the function packet_send2() in packet.c will buffer any outgoing messages with types outside of the range 1-49, and only sent these later once re-keying has been completed (by returning to state CLIENT_REKEX in Fig. 7). This assures that no outgoing messages of higher protocol layers (marked as traffic_C in Fig. 7) will be sent during re-keying, in accordance with Fig. 7 and the comment to that effect at the bottom of page 19 in [SSH-TRANS].

9.2 The server

The server (in the basic mode) sets up a listening socket in server_listen() (called from main() in sshd.c) and then the incoming clients are accepted in server_accept_loop() (also called from main()). Each time a client connects, the server process forks to handle the client in a sub-process. This sub-process returns from server_accept_loop(). The parent process that listens for new incoming clients never returns from server_accept_loop().

The main() procedure for the server (daemon) in sshd.c

- calls the procedure sshd_exchange_identification() takes care of exchanging version numbers: it sends the server's identification string and then waits for the client's. (VERSION$_S$! VERSION$_C$!)
- then calls do_ssh2_kex() to do the key exchange, discussed below;
- then calls do_authentication2() for the user authentication.
- finally calls do_authenticated() to handle the session, which calls down to client_loop2().

For the key exchange, do_ssh2_kex() in sshd.c works rather like ssh_kex2 for the client:

- it defines a Kex structure kex with function pointers to kexdh_server() and kexgex_server() as procedures to use for the key exchange;
- calls kex_setup(), which in turn – just as for the client – (KEXINIT$_C$!)
  - calls kex_send_kexinit() to send SSH2_MSG_KEXINIT;
  - calls kex_reset_dispatch() to reset the dispatch table, setting all transport protocol messages (the range 1- 49) except KEX_INIT to be treated as errors;
- calls dispatch_run.

(KEXINIT$_S$?) Now dispatch_run() will only respond to KEXINIT , and hand over control to kexdh_server() or kexgex_server() for the actual key exchange.

The procedure kexdh_server()

- (KEXDH_INIT?) waits for a KEXDH_INIT from the client (using packet_read_expect());
- (KEXDH_REPLY!) replies with its KEXDH_REPLY message;
- (NEWKEYS$_S$! NEWKEYS$_C$?) calls kex_finish() which first sends NEWKEYS and then receives SSH2_MSG_NEWKEYS using packet_read_expect.

The procedure kexgex_server

- (GEX_REQUEST?) waits for a KEX_DH_GEX_REQUEST or KEXDH_GEX_REQUEST_OLD message, using packet_read(), aborting the protocol with a fatal error if any other message arrives; packet_read() calls down to packet_read_seqnr(), which takes case of the generic messages in the correct way;
- (GEX_INIT?) waits for a KEX_DH_GEX_INIT from the client (using packet_read_expect);
- (GEX_REPLY!) replies with its KEXDH_GEX_REPLY message
- (NEWKEYS$_S$! NEWKEYS$_C$?) calls kex_finish() which first sends NEWKEYS and then receives SSH2_MSG_NEWKEYS using packet_read_expect.

So, as far as the incoming messages for transport layer protocol are concerned, the server handles KEXDH_INIT and NEWKEYS with packet_read_expect. Only an incoming KEX_INIT from the client will be handled via the dispatch table.

9.3 Initialising and resetting of the dispatch table

The dispatch table is not explicitly initialised when the client or server starts. As `dispatch` is a global array, the ANSI C standard guarantees it is initialised with NULLs[6]. Still, for clarity and graceful degradation – with an error message rather than simply crashing – it would be nice to fill the dispatch table with error handling procedures in the beginning, by calling

```
dispatch_init(&dispatch_protocol_error);
```

After the transport layer protocol and the authentication protocol have been run, just before the interactive session starts, both client and server re-initialise their dispatch table. The client does this by invoking `client_init_dispatch()` from `clientloop()` in `clientloop.c`, the server does it by invoking `server_init_dispatch()` from `serverloop2()` in `serverloop.c`.

Both `client_init_dispatch()` and `server_init_dispatch()` reset all entries in the dispatch table to `dispatch_protocol_error`, except

- one transport layer message, namely `KEXINIT`,
- all generic connection messages (numbers 80-82), and
- all channel related messages (number 90-100).

The entry in the dispatch table for `KEXINIT` is set to `kex_input_kexinit()` to restart the key exchange when the keys are refreshed.

During the transport layer protocol both client and server reset the dispatch table by calling `kex_reset_dispatch()`

- at the beginning of a new key exchange, directly after sending their `KEX_INIT` message[7],
- at the end of a key exchange, just before sending their `NEWKEYS` message[8]

The procedure `kex_reset_dispatch()` resets the dispatch table such that all transport protocol messages (i.e. messages types 1-49) *except* `KEX_INIT` are treated as errors, setting the corresponding entries to `kex_protocol_error()`.

Note that resetting the dispatch table at the end of the key exchange is redundant, as none of the entries of the dispatch table are changed during key exchange.

In fact, for all transport protocol messages (i.e. message types 1- 49) except `KEX_INIT` the corresponding entries in the dispatch table are only ever set to `kex_protocol_error()` or to `dispatch_protocol_error()`. The only difference between these two error procedures is that the former reports "a kex protocol error" and the latter "a dispatch_protocol_error". Apart from `KEX_INIT`, all transport protocol messages are processed by `packet_read_expect`, and never via the dispatch table, so the dispatch table can indeed treat these messages as errors.

9.4 Discussion

The OpenSSH client does not take advantage of the possibility of optimistically sending the end first key exchange message directly after sending its `KEX_INIT`. Instead, it always waits for the incoming `KEX_INIT` first.

The server does not do this either, but then the server could not do this, as the first key exchange message has to be sent by the client for all key exchange method supported by OpenSSH.

The client is not prepared to accept and ignore an initial key exchange message from the server - this would lead to a disconnection. However, this situation is not likely – or even possible – as none of the existing key exchange methods (which are all variants of Diffie-Hellman) would allow a server to optimistically send a first key exchange message.

The OpenSSH implementation does not quite conform to the following requirement [SSH-TRANS, §11.4]

[6] Thanks to Darren Tucker for pointing this out.

[7] To be precise, this happens in `kex_setup()`, which the client calls in `ssh_kex2()` and the server calls in `do_ssh2()`.

[8] To be precise, this happens in `kex_finish()`, which is called from `kexdh_server`, `kexdh_client`, `kexgex_client`, and `kexgex_server`.

```
An implementation MUST respond to all unrecognised messages with an
SSH_MSG_UNIMPLEMENTED message in the order in which the messages were received.
Such messages MUST be otherwise ignored. Later protocol versions may define
other meanings for these message types.
```

Such an `SSH_MSG_UNIMPLEMENTED` messages is sent if an unrecognised incoming message is handled via `dispatch_run()`, but not if it is handled by `packet_read_expect()`.

10 Related Work

Research into the analysis of security protocols is gradually beginning to tackle the problem of looking at actual source code instead of more abstract representations of security protocols.

Closest to our work in spirit here is the work of Udrea et. al. [23] on using a static analysis of C source code to check if it obeys constraints on the ordering of operations and on data values. The constraints on the ordering of operations is precisely what we try to capture in our state machines. The approach has been tried on SSH, where the authors extracted 87 rules from the RFC.

Apart from the fact that the approach is backed up by a static analysis tool, the approach of Udrea et. al. is more ambitious than ours when it comes to describing the protocol, in that it tries to do more than just capture the order of operations, as we do, by also imposing constraints on the message contents.

Describing the protocol as a set of constraints is closer to the style used in the RFCs than our state-based approach, but more likely to be only a partial specification, and underspecify the set of allowed interactions. Our state-diagrams provide a more precise description of the order of operations and are in fact likely to overspecify, especially the later versions which choose a particular resolution of the possible concurrency.

More ambitious efforts to analyse C source code of security protocols [7, 12] are more semantical in nature, and try to prove security of the protocol implementation, not just conformance to some protocol specification (in the form of a state-diagram or a set of constraints). This involves modelling of an attacker and symbolically determining the knowledge an attacker might be able to collect, and then proving security properties by model-checking [7] or automated theorem proving [12]. Unfortunately, applying these techniques to a C implementation as complex as OpenSSH, with e.g. its extensive use of function pointers, does not seem feasible yet.

Fournet and Gordon and their co-workers has been developing techniques to verify security properties of protocols implemented in functional languages, initially by translating ML-like programs to ProVerif, a resolution-based theorem prover for cryptographic protocols [5], but more recently using refinement types for a variant of F# [3, 4] and automated theorem provers to discharge the proof obligations that arise as part of type checking. Of course, F# is a much cleaner programming language than C.

An analysis as we have done for SSH here has been repeated for TLS [20]. The specification of TLS seems to be a lot more structured than that of SSH, and the state diagram much easier to obtain from the specification. An informal code review of the TLS implementation studied there revealed some deviations from the spec, but not in ways to would compromise security.

11 Conclusions

We presented partial but rigorous specifications of the SSH transport layer, and discussed a code review of OpenSSH using these specifications. Our specifications are partial in that they only consider the (dis)allowed orders of the different messages types in legal protocol runs. The specifications could also be used for more formal, tool-supported analysis of the code, as demonstrated in [17] for a Java implementation of SSH, but we do not know of techniques that could cope with the C implementation of OpenSSH to do this.

We believe that a thorough code review of OpenSSH is impossible without effectively doing the work that we have done in analysing the RFCs. After all, any implementation of SSH will somehow implement a state machine that should conform to the RFCs specifying the protocol. Providing rigorous specs in the form of state diagrams is therefore a useful first step for any code review. Of course, it would also be a useful first step for developing any implementation.

It is a pity that the state diagrams hiding in RFCs that specify SSH are so implicit. Making them more explicit would make the specification more useful. The fact that RFCs are produced in ASCII is not conducive to including state diagrams, of course. It is interesting to note that the RFC specifying FTP [19] includes state diagrams as ASCII art. We conjecture that in many other settings the useful state diagrams that people draw on whiteboards and napkins ultimately don't make it to the official documentation, which seems a pity.

References

1. S. Andova, C. Cremers, K. Gjøsteen, S. Mauw, S.F. Mjølsnes, and S. Radomirović. A framework for compositional verification of security protocols. *Inf. Comput.*, 206(2-4):425–459, 2008.
2. M. Backes, S. Lorenz, Maffei M, and K. Pecina. The CASPA tool: Causality-based abstraction for security protocol analysis. In *CAV'08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 419–422. Springer, 2008.
3. J. Bengtson, K. Bhargavan, C. Fournet, A.D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF 2008*, pages 17–32. IEEE, 2008.
4. K. Bhargavan, C. Fournet, and A.D. Gordon. Modular verification of security protocol code by typing. *ACM SIGPLAN Notices*, 45(1):445–456, 2010.
5. K. Bhargavan, C. Fournet, A.D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(1):1–61, 2008.
6. B. Blanchet. Automatic verification of correspondences for security protocols. *J. Comput. Secur.*, 17(4):363–434, 2009.
7. S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *IEEE Computer Security Foundations Symposium*, pages 172–185, 2009.
8. C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Ph.D. dissertation, Eindhoven University of Technology, 2006.
9. S.F. Doghmi, J.D. Guttman, and F.J. Thayer. Searching for shapes in cryptographic protocols. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007*, volume 4424 of *LNCS*, pages 523–537. Springer, 2007.
10. J. Engelhardt. Detecting and deceiving network scans. Available from `http://jengelh.medozas.de/documents/Chaostables.pdf`, 2007.
11. M. Friedl, N. Provos, and W. Simpson. Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol. RFC 4419, The Internet Engineering Task Force, Network Working Group, 2006.
12. J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI'2005*, volume 3385 of *LNCS*, pages 363–379. Springer, 2005.
13. B. Harris. Rivest-Shamir-Adleman (RSA) key exchange for the Secure Shell (SSH) Transport Layer Protocol. RFC 4432, The Internet Engineering Task Force, Network Working Group, 2006.
14. E. Hubbers, M. Oostdijk, and E. Poll. From finite state machines to provably correct Java Card applets. In *Proceedings of the 18th IFIP Information Security Conference, Athens, Greece*, pages 465–470. Kluwer Academic Publishers, 2003.
15. J. Hutzelman, J. Salowey, and J. Galbraith. Generic Security Service Application Program Interface (GSS-API) Authentication and Key Exchange for the Secure Shell (SSH) Protocol. RFC 4462, The Internet Engineering Task Force, Network Working Group, 2006.
16. S. Lehtinen. The Secure Shell (SSH) Protocol Assigned Numbers. RFC 4250, The Internet Engineering Task Force, Network Working Group, January 2006.
17. E. Poll and A. Schubert. Verifying an implementation of SSH. In R. Focardi, editor, *WITS'2007*, pages 164–177, 2007.
18. J. Postel. Internet Protocol. RFC 791, The Internet Engineering Task Force, Network Working Group, 1981.
19. J. Postel and J. Reynolds. File Transfer Protocol (FTP). RFC 959, The Internet Engineering Task Force, Network Working Group, January 1985.

20. P. Rogaar. Security analysis of a TLS implementation using finite state machines, 2010. Unpublished manuscript.
21. D. Xiaodong Song. Athena: a new efficient automatic checker for security protocol analysis. In *CSFW'99: Proceedings of the 12th IEEE workshop on Computer Security Foundations*, page 192, Washington, DC, USA, 1999. IEEE Computer Society.
22. D. Stebila and J. Green. Elliptic-Curve Algorithm Integration in the Secure Shell Transport Layer. RFC 5656, The Internet Engineering Task Force, Network Working Group, 2009.
23. O. Udrea, C. Lumezanu, and J.S. Foster. Rule-based static analysis of network protocol implementations. *Information and Computation*, 206(2-4):130–157, 2007.
24. L. Viganò. Automated security protocol analysis with the avispa tool. *Electr. Notes Theor. Comput. Sci.*, 155:61–86, 2006.
25. D. von Oheimb. Formal specification of the SSH transport layer protocol in HLPSL, 2004. Available online at `http://www.avispa-project.org/library/ssh-transport.html`.
26. T. Ylönen. The Secure Shell (SSH) Authentication Protocol. RFC 4252, The Internet Engineering Task Force, Network Working Group, January 2006.
27. T. Ylönen. The Secure Shell (SSH) Connection Protocol. RFC 4254, The Internet Engineering Task Force, Network Working Group, 2006.
28. T. Ylönen. The Secure Shell (SSH) Protocol Architecture. RFC 4251, The Internet Engineering Task Force, Network Working Group, January 2006.
29. T. Ylönen. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, The Internet Engineering Task Force, Network Working Group, January 2006.