Radboud University Nijmegen

Bachelor Thesis

Evaluating implementations of SSH by means of model-based testing

Author:
Erik Boss

Supervisor:
Erik Poll

Abstract

This thesis presents an evaluation of implementations of the Secure Shell (SSH) protocol by applying model-based testing techniques. In doing so, an evaluation can be given of the techniques themselves when applied to a fairly complex system such as SSH. More specifically, only the server side of the OpenSSH implementation of the SSH transport layer is actually tested and evaluated. However, this can be easily extended to test the client side and/or other implementations. The process of testing and evalution is detailed in order to provide a guide to *a*) testing the rest of SSH; *b*) applying model-based testing techniques to similar protocols as SSH.

August 6, 2012

Contents

1. Introduction

The Secure Shell (SSH) protocol and the tools built upon it such as *scp* and *sftp* are often vital for the tech savvy user. It is therefore important that an implementation of SSH implements the specification correctly. Model-based testing techniques claim that they can do just that, testing whether a specification is implemented correctly.

This thesis aims to test an implementation of the SSH Transport Layer, OpenSSH in this case, with model-based testing techniques and in doing so give an argument for the (in)correctness and (in)security of this implementation. Also, it will propose and implement a way to work around some of the problems encountered when preparing an application for model-based testing. A deliberate effort was made to make the solutions proposed work at least for other parts and/or implementations of (Open)SSH. At best, the solutions will help with the testing of implementations of other protocols or systems as well. Lastly, an evaluation is provided of both the implementation of the SSH transport layer and the model-based testing process in general.

Note that this thesis does not aim to build and test models of all the different levels and protocols of the SSH architecture. Also, this thesis does not give a proof that the implementation of SSH that is tested is either secure or correct.

Section 2 will provide some of the background of this thesis, i.e. some information about model-based testing, SSH and the tools used. Section 3 contains the basic models of the SSH Transport Layer, as proposed in earlier research. The bulk of this thesis is in section 4, which describes both the preparation needed for testing and the testing itself. Section 5 evaluates OpenSSH on the basis of the test results. Furthermore, it attempts to evaluate the usefulness of model-based testing of complex systems such as SSH.

2. Background

To understand the later sections properly some background knowledge is convenient and so this section will provide some information about the most important notions like SSH and model-based testing. Also, it provides information about the tools used in the process of testing.

2.1. Model-based Testing

Model-based testing is a method of testing the correctness of a given system or component by means of executing testcases that are generated systematically from a model of the system[6]. Typically, this type of testing deals with testing functionality of the System Under Test (SUT) or Implementation under Test (UIT) as a black box. This process is illustrated in Figure 1. It consists of the following components:

Model: A model of correct and/or desired behaviour created on the basis of a specification of the SUT. This model usually takes the form of some sort of finite automaton such as regular Deterministic Finite Automata (DFA's) or Mealy Machines, as used in [5].

Test application: Some machine that systematically derives tests from the model. These tests typically consist of series of inputs and their corresponding outputs.

Test driver: The part of the test application that communicates with the SUT by means of the abstract input and output. These will often take the form of some sort of label. For instance, in Figure 3b there is a label `?VERSION_C` for inputting the client's version exchange message. Likewise, abstract output can be something like `!VERSION_S` which means that the server has sent its version exchange message to the client.

Adapter: The adapter is in charge of converting abstract input to concrete input and concrete output to abstract output. This first conversion typically entails converting a label into something the SUT understands. This can be function call, a message sent over a socket, etc. The adapter is not needed if the abstract input/output and concrete input/output are the same. The result of which is that the test driver and SUT can communicate directly.

System Under Test: The system being tested. This can treat it as a black box, we can only know of behaviour that is actually externally observable.

Model-based testing is particularly useful for those cases where one has limited or no access to the source code of the implementation. Of course, it can also be used when the source code is available.

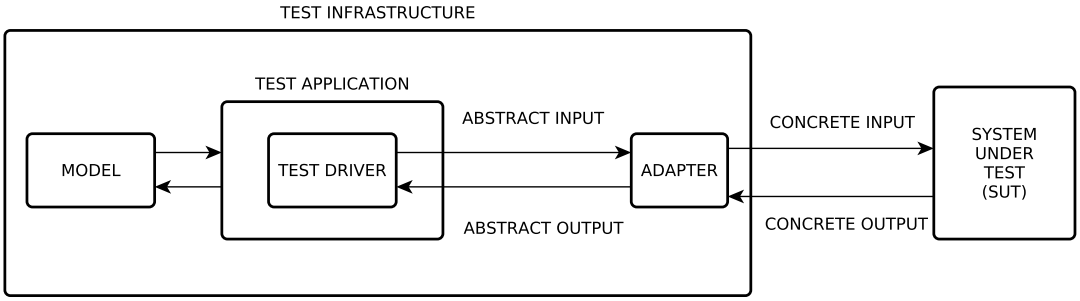


Figure 1: Model-based testing with an adapter.

Several tools exist to assist one in using model-based testing techniques to test a given system. One such tool is *JTorX*[1], which is a tool for deriving model-driven tests and executing those tests.

2.2. Secure Shell Protocol

The Secure Shell (SSH) protocol is a *protocol for secure remote login and other secure network services over an insecure network*[11]. The protocol is specified in five so-called *Request for Comments* documents (RFC's), namely [10],[11],[12],[13] and [14], hereafter ref-

erenced respectively as [SSH-NUMBERS], [SSH-ARCH], [SSH-USERAUTH], [SSH-TRANS] and [SSH-CONNECT]. This is the same convention as used in the RFC's.

The RFC's contain the following information:

[SSH-NUMBERS] denotes the numbers and labels used in the protocol for use in, for instance, error messages.

[SSH-ARCH] contains a description of the architecture used, the security goals it wants to reach and the terminology and notations used in the RFC's.

[SSH-USERAUTH] describes the authentication protocol and public key, password and host-based client authentication methods.

[SSH-TRANS] describes the SSH transport layer protocol, Diffie-Helman key exchange and the minimal amount of algorithms needed to implement the SSH transport layer protocol.

[SSH-CONNECT] describes the SSH Connection protocol for, amongst others, interactive login sessions and X11 forwarding.

SSH is most commonly used for securely accessing shell accounts on UNIX-like systems. Several versions of the protocol are used today, but for the rest of this document the use of SSH-2 (as opposed to SSH-1) is assumed, when the particular version is not explicitly specified.

[SSH-TRANS] describes, amongst other things, the Binary Packet Protocol (BDP). BDP is a specific format for packets used throughout the SSH protocols. With the execption of a few messages in the beginning of the transport protocol, all communication is done with these BDP packets. Key amongst the information stored in the packet is the message type. This is one of the types defined in [SSH-NUMBERS], for instance SSH_MSG_KEXINIT. Note that in further usage of these message labels, the prefix SSH_MSG will be left out. The whole packet, as stated in [SSH-TRANS], is this:

```
uint32    packet_length
      byte      padding_length
      byte[n1]  payload; n1 = packet_length - padding_length - 1
      byte[n2]  random padding; n2 = padding_length
      byte[m]   mac (Message Authentication Code - MAC); m = mac_length
```

2.3. OpenSSH

The most common implementation of SSH is, at the time of writing, *OpenSSH* [2] which aims to be a

> (...) FREE version of the SSH connectivity tools that technical users of the Internet rely on. Users of telnet, rlogin, and ftp may not realize that their password is transmitted across the Internet unencrypted, but it is. OpenSSH

encrypts all traffic (including passwords) to effectively eliminate eavesdropping, connection hijacking, and other attacks. Additionally, OpenSSH provides secure tunneling capabilities and several authentication methods, and supports all SSH protocol versions.

Originally, OpenSSH was developed by and for the OpenBSD Project[3] but since its introduction versions for other operating systems have been developed. It is written in C and actively maintained. The most recent stable version of the Linux version as of June 2012 (the one used for this thesis) is `v6.0p1`, released on the 21st of April 2012.

2.4. Tools

A brief overview of the tools used in the process of testing SSH. JTorX serves as the test application, yEd was used for the creation of the models.

2.4.1. JTorX

JtorX is a a tool for model-driven test derivation and execution, based on the ioco theory[1]. It serves as the testing tool for the model-based testing done in this thesis. It was chosen due to its relative easy-of-use for people not accustomed to such methods and its support for various formats of model specifiction. The theory on which it builds is neatly explained in [6] and I will forego repeating that information here.

2.5. yEd

yEd[4] is java-based cross-platform graph editor. Notable is the fact that it supports exporting to `graphml` files, and this format happens to be one of the formats JTorX accepts. This allows us to easily create the models needed for model-based testing without having to learn a graph specification language.

3. Models of SSH and OpenSSH

In order to test any implementation of SSH, including OpenSSH, by means of model-based testing techniques we obviously need a model to test with. In our case, since we are interested in the SSH transport protocol, we need a model that fully specifies the protocol as given in the RFC's, most notably in `[SSH-TRANS]`.

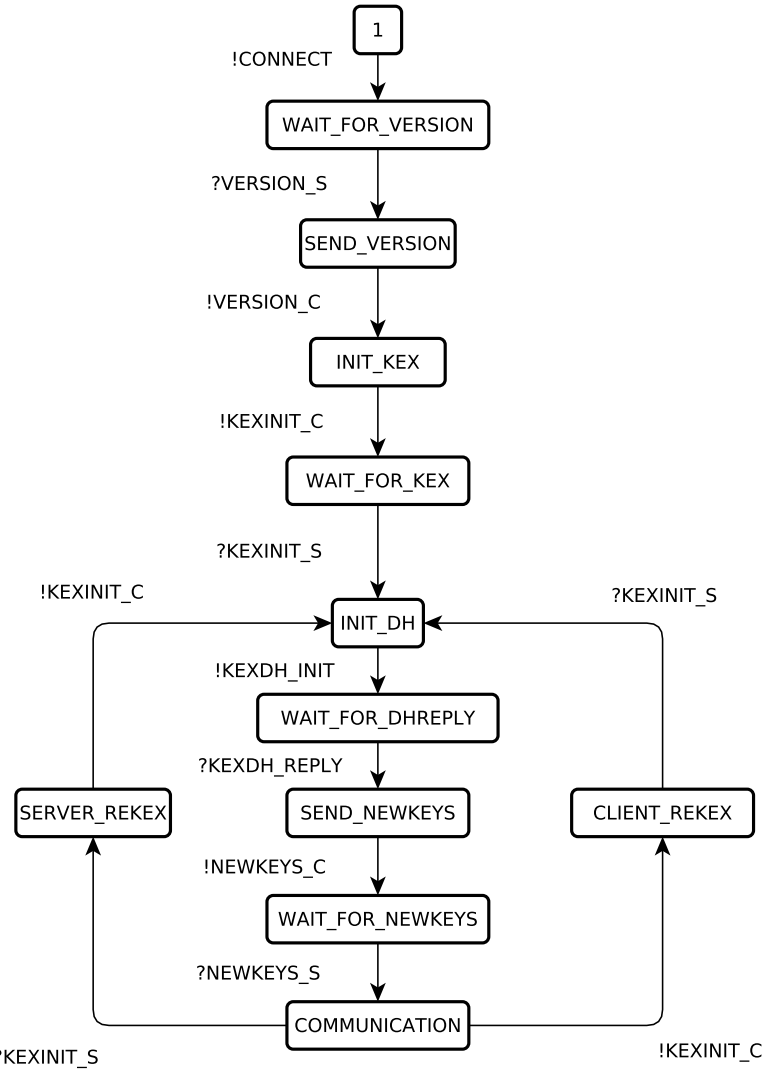
Creating a model of SSH is rather difficult since the RFC's don't explicitly contain this model. So one has to derive this model from the specification and the model is therefore prone to errors of interpretation. Luckily, this work has already been done. In [8] and [7] we find models respectively for the OpenSSH and SSH Tranport layer. Since redoing all the work is unnecessary, these are the models of SSH we will consider. Note that some changes have been made to these models. Some labels on states and/or edges were renamed in order to make the models more alike and to make it easier to write a generic adapter later on. Also, the first model has had extra states added to make sure only one label exists for a given edge. Lastly, the `TRAFFIC` transitions were removed

since they only appeared in one of the models and since they are part of the higher level protocols they are not needed for the transport layer models. Also note that both of these models only describe the expected behaviour of the implementation and thus they are referred to as being *benign*. Later on, *malicious* versions will be introduced that try to break the implementation.

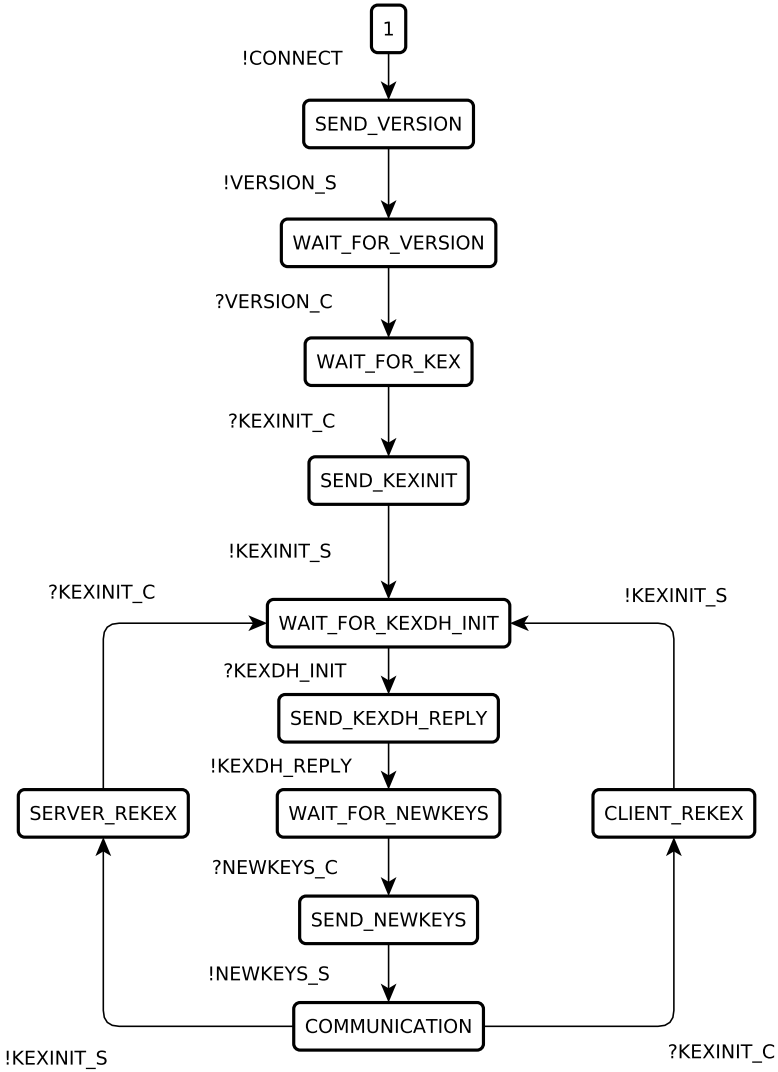
The reason why two different models are given will be obvious later on. The model of SSH will be used first in testing, and this testing will reveal that the implementation of OpenSSH is indeed more similar to the OpenSSH model. This makes sense, since that model was crafted, at least partially, on the basis of the implementation, as opposed the being based on the specification only.

The models in Figure 2 and Figure 3 don't differ too much if you look carefully. The most prominent difference is the fact that the SSH protocol in general allows for some parallelism in the messages sent. For instance, there is no fixed order in who (server or client) sends their version, `VERSION_S` and `VERSION_C`, first. The model for OpenSSH does impose such an order. For OpenSSH, the server always sends its `VERSION_S` first.

The `KEXINIT` and `NEWKEYS` messages are a different sort of beast. In the models of OpenSSH, Figure 2, it appears as though the client sends its message first. This is not entirely the case. Both client and server send their respective `KEXINIT` messages first and then read the other party's `KEXINIT`. This was described, although not explicitly, in [8], and can be verified by analysing the order of function calls in `ssh_kex2()` and `do_ssh2_kex()`. So in this, the SSH model is more accurate than the OpenSSH model since it tries to model this. That is, it matches the description in the specification *and* it matches the actual implementation, However, we will see that during testing it appears as though the client sends its message first. The same thing holds for `NEWKEYS` as is apparent from `kexecdh_client()` and `kexecdh_server()` since they both call `kex_finish()` which first sends the `NEWKEYS` packet and then reads it.

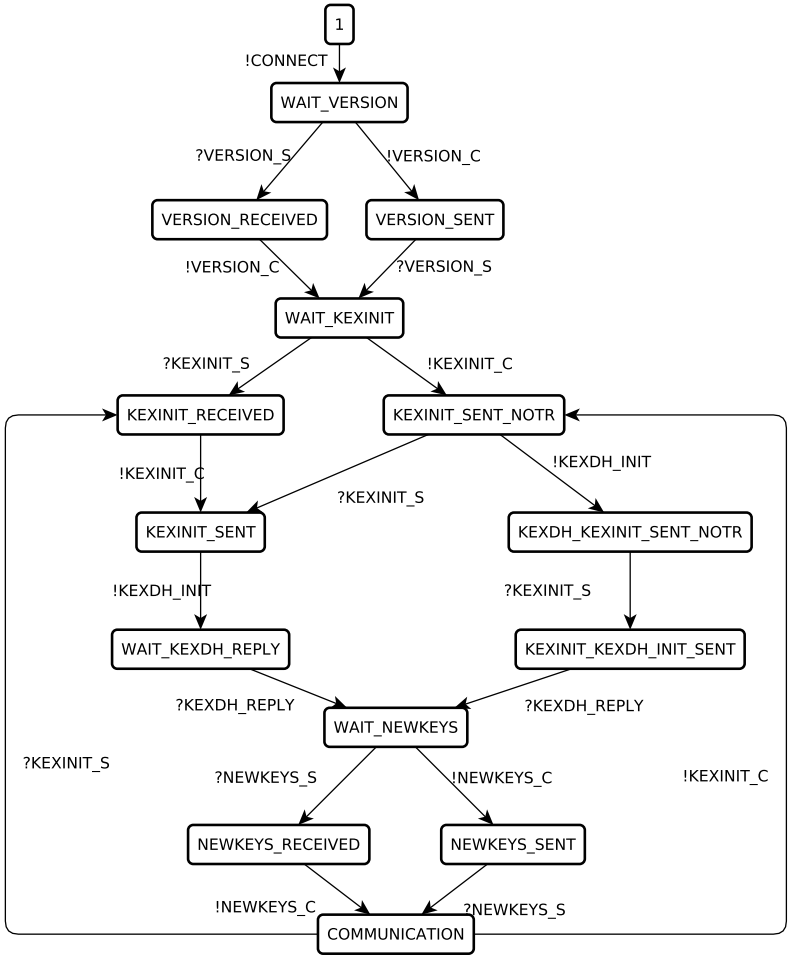


(a) Client

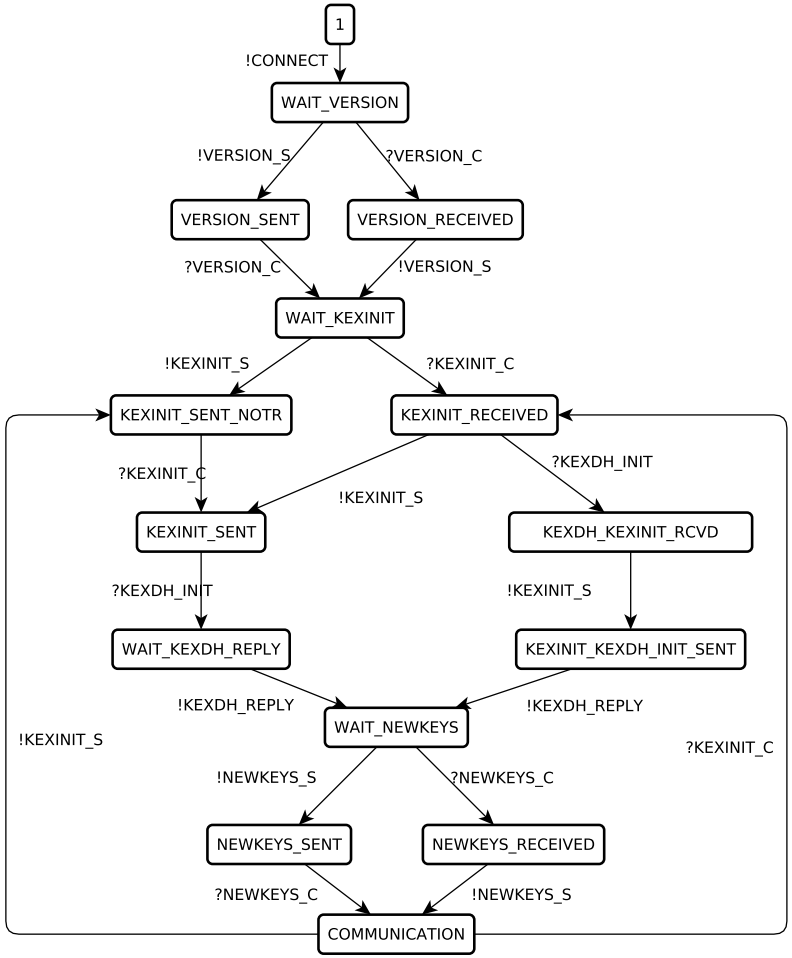


(b) Server

Figure 2: Models of OpenSSH (benign)



(a) Client



(b) Server

Figure 3: Models of SSH with parallelism (benign)

4. Testing implementations of SSH

In this section the development and design of the testing setup is described. Also, the results of the testing are presented.

4.1. Setup

The goal of the setup is to prepare a test architecture that allows for the model-based testing of *any* SSH server implementation. Figure 1 showed a general setup for model-based testing. If we were to implement this directly for the SSH Transport layer on the server side, this would result in something like in Figure 4. This is a perfectly fine way of doing things. However, there are a few problems. The easiest way to send packets from the adapter to the server is by calling functions in the client. However, the client of choice, OpenSSH, keeps a lot of state. The prime example of this is the dispatch table, an array of function pointers that contains different values in different stages of the protocol. Also, the functions are full of side effects that edit this state. This makes it very hard to send packets to the SUT.

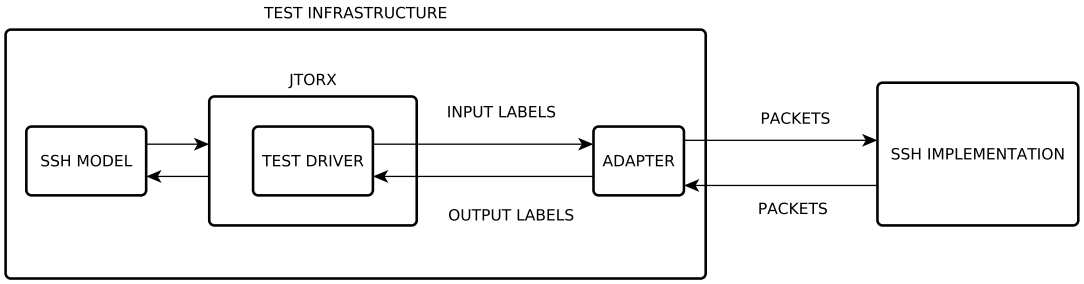


Figure 4: Model-based testing of SSH with an adapter.

Considering this, another approach was needed. The idea is simple: instead of keeping the adapter and the OpSSH client separate or perhaps trying to create the packets manually, extend the OpenSSH client with the functionality of the adapter. The client will keep the state correct for the extension to call functions in the client in order to send packets to the server. The client should be modified to only send packets when the extension allows it. This combination of extension and client will hereafter be known as the extended adapter.

Now the test driver can apply stimuli to the extended adapter, as though the above changes never happened, and in doing so test the server. It is very important that care is taken to not interfere with the client too much. In order to preserve the validity of the client's function calls and state, the changes made should not be more obtrusive than necessary.

As it turned out, the extended adapter was sending quite a bit more output to the driver than expected. Ideally, we want it to only communicate via labels (as in Figure 4)

and due to the extra output this was initially not the case. Two solutions come to mind: filtering the output and/or turning some of the output off. The former seems the better solution since we want to avoid changing/extending the client as much as possible. Filtering can be done outside the SUT and does therefore require no modifications to the client. Furthermore, we also needed something to start the actual OpenSSH process and be able actually send labels to this process. A solution could be to combine all these things into a component separate from the adapter. This component would then serve as a kind of *middleman* between the adapter and the test driver.

Given this middleman, the whole scheme would then look like Figure 5. Note that the middleman is optional in the sense that is useful for testing SSH but it may not be needed for testing implementations of other protocols. Also, this functionality can also be directly implemented in the extended adapter. This would, however, require further modifications to the client and this should be avoided where possible.

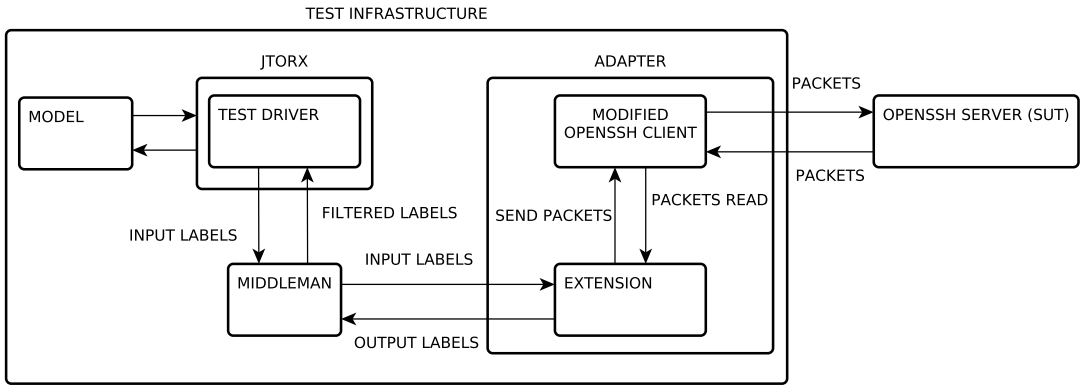


Figure 5: Model-based testing with a middleman and extended adapter.

As a side note, this approach, with an adapter integrated in some client and output filtering with a separate component, can be used for testing other protocols as well, provided that the client is modifiable.

4.2. Implementing the middleman

One of the advantages about separating the middleman from the adapter is that is does not necessarily have to be written in the same language as the SUT. The same is often true for the adapter, but for the adapter this is much more difficult since it probably needs to call functions on the SUT.

Because of the freedom to choose a different language and the fact that filtering input is relatively hard in C (OpenSSH's implementation language), the middleman was written in Perl. Perl was specifically chosen since it has both great text processing power and a nice interface to processes on a UNIX-like system.

What `middleman.pl` (for the source, see appendix A) does is actually rather simple. It:

1. Spawns a OpenSSH process with the adapter integrated;
2. Determines which file handles are open for reading;
3. Reads the input from each open file handle, thus making sure that the input and output are interleaved (if both filehandles are open);
4. Filters the output by checking whether the output consists of a message the adapter recognizes.
5. Writes the filtered label(s) from the extended SSH process to the driver;
6. Writes the label(s) from the driver to the SSH process;;
7. Repeat from 2 until the test driver is done testing or until the SSH process terminates

The filtering is very simple. For instance, say the adapter outputs `!VERSION_S`, this is a label that the test driver (JTorX) will recognize since it is a valid transition label in the model. However, if the adapter outputs `[someuser@hostname ~]$` (which is the PS1 or prompt) then this should not be written to the test driver since it is not a label that it recognizes.

4.3. Implementing the adapter

The extended adapter, i.e. the modified OpenSSH client with an extension, as it is implemented is meant to do one thing and one thing only. It attempts to make the OpenSSH client input enabled, which it is not by default. Normally this means listening for input and calling for functions corresponding to the input. The same holds here, except that the implementation is different. For one, in order to keep the state valid, the adapter only reacts on input when the client actually wants to do something relevant. That means, the extension reacts when the client wants to send a packet to the server. This ensures, as much as possible, the validity of the state whenever the adapters reacts. At any other time this cannot be guaranteed.

When the the client tries to send a packet the following happens:

1. If the packet is not part of the SSH transport layer it is sent along to the server as if nothing happened. If it is, the adapter looks for output from the test driver (or the middleman, to be precise).
2. If the label received from the test driver is the same as the label corresponding with the type of the packet, the packet is sent to the server unaltered. If it is not the same, the packet that corresponds to the output from the test driver will be send to the SUT. This typically entails calling some function(s) that actually take care of sending the packet.

Besides being able to react to input from the test driver, the adapter also has to be capable of observing output from the SUT. Again, this output corresponds to the transitions in the model with a '!' prepended. Luckily, this is very easy. When a packet is read, the `packet_read()` family of functions return the *type* of the packet read. Since these packets are the output from the server, the type only has to be converted to a label and then printed to the test driver by the adapter. Due to the fact that the middleman filters the output it doesn't matter whether the packet is part of the transport layer or not.

Some implementation details were obviously left out. For a better idea what had to change to make all the above work, see Appendix B, Appendix C and Appendix D. In these files, the only changes are those in a `#ifdef USE_ADAPTER` block.

4.4. Testing environment

For the testing both the server and the extended client were running on the same laptop, in this case a Lenovo X220 with 4GB RAM and an Intel Core i5 processor. The laptop was running the most recent version of Arch Linux at that time[1]. Running everything on the same machine allowed for easy monitoring of both the server (sshd) and client (ssh) processes. The OpenSSH source code for client and server are from version 6.0p1, the most recent stable version dating from the 21st of April 2012 as of June 2012.

A SSH keypair was created to facilitate passwordless login to the SSH server. This made sure that the only interactions with the client were those required for using the adapter. The middleman could have been adapted to handle login with a password, but this is not as convenient.

4.5. Testing the OpenSSH Server

As stated earlier, for the testing of the OpenSSH server implementation JTorX was utilized. Configuring JTorX is rather easy and given the fact that start out with the begin SSH server model (Figure 3b), setting JTorX up is then a simple matter of providing the path to the middleman and the path to the model in graphml format. After this, the actual testing is rather straightforward and goes as described in subsubsection 2.4.1.

4.6. Acceptance

First, I had JTorX run tests to see whether OpenSSH provides the behaviour specified in the benign SSH server model. Due to the benign SSH server model only containing correct behaviour, this is an acceptance test. Recall that this model does not contain information on how unexpected messages are handled. These unexpected messages will be handled in the next section. As expected, this test failed. This was, of course, due to the fact that the model has the parallelism we touched upon earlier and OpenSSH does not really implement this. The model contains transitions and states that do not occur when testing. I found that:

[1] Arch Linux is a rolling-release distribution and has no version number for easy referencing.

- VERSION_S is always sent before VERSION_C.
- KEXINIT_C always takes place before KEXINIT_S. Do note that this is not precisely true due to the reasons described in section 3.
- KEXDH_INIT is always sent after both parties have sent and received their KEXINIT messages.
- NEWKEYS_C is always sent before NEWKEYS_S. Again, this is due to the same reasons as held for KEXINIT.
- The server did not take initiative in asking for a re-keying procedure.

Of course, if the model in Figure 3b were to be changed to accommodate these differences, the model would be the exact same model given by Figure 2b. Indeed, when this model was loaded into JTorX and given the same treatment, it passed the tests admirably except for the last point.

The problem with this last point is that it does happen, but only after an hour or so. A single test run, when performed automatically, takes seconds, not hours. [SSH-TRANS] has the following to say about this:

> It is RECOMMENDED that the keys be changed after each gigabyte of transmitted data or after each hour of connection time, whichever comes sooner. However, since the re-exchange is a public key operation, it requires a fair amount of processing power and should not be performed too often.

This notion is implemented in packet.c as packet_need_rekeying(). It seems reasonable that the conditions here actually do occur but I haven't yet been able to force them in a test environment. For this reason I will not remove the transition from the model.

Acceptance testing yields a model that is highly similar to the model proposed in [8] and seen in Figure 2b. We will therefore use this model to do the tests in the next section.

4.7. Security

In order to make use of JTorX in testing for unexpected input and/or output we need to make a few changes to the model. Intuitively, this results in an adapter that can also send messages in the wrong order. To do this, We add a new state UNDEFINED_STATE (see Figure 6) and create transitions for every possible input or output message that we recognize to itself (UNDEFINED_STATE). Also we create transitions from every state that gets input to this undefined state. This new model is the *malicious* OpenSSH server model. For example, a transition is made from WAIT_FOR_VERSION, but not from SEND_VERSION. We label these transitions with τ or tau which signifies a non-observable transition. A quick test shows that this malicious model still accepts all the series of

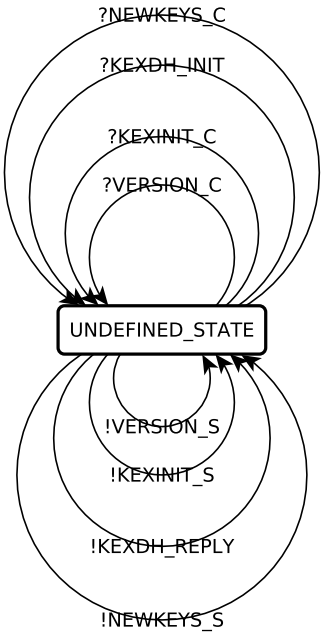


Figure 6: The undefined state.

input/output that the earlier model did. This makes sense because this new model actually contains all traces that consist of the messages contained in `UNDEFINED_STATE`. Formally, the benign OpenSSH server model is a strict subset of the malicious OpenSSH server model.

State / Message	`VERSION_C`	`KEXINIT_C`	`KEXDH_INIT`	`NEWKEYS_C`
`WAIT_VERSION`	M	$\perp$	$\perp$	$\perp$
`WAIT_FOR_KEX`	$\perp$	$\perp$	M	$\perp$
`WAIT_FOR_KEXDH_INIT`	$\perp$	$\perp$	M	$\perp$
`WAIT_FOR_NEWKEYS`	$\perp$	$\perp$	$\perp$	M
`COMMUNICATION`	$\perp$	M	$\perp$	$\perp$
`SERVER_REKEX`	$\perp$	$\perp$	M	$\perp$

Table 1: Results of (un)expected input. $\perp$ signifies termination of the SUT, M signifies that this transition was part of the benign OpenSSH model and so is not considered unexpected input.

Now we have access to all stimuli and observations in every state. This allows us to determine the effects of sending unexpected input. The effects obtained are stated Table 1.

The SUT terminates when given unexpected input due to the following reasons:

- Sending a proper BDP, as defined in subsection 2.2, packet before the versions are

exchanged can not work due to uninitialized variables (especially the key struct). Also, the server is expecting a raw version string to be sent and any of the other packets are in no way similar to this version string.

- Sending a version string is, as said before, just the sending of raw data and therefore if the server thinks (due to the state of the protocol) that it is supposed to read a BDP packet, it will terminate. Note that it only terminates after the next BDP packet is sent, since that is when it notices the corruption.
- Everything that relies on keys (all except for `VERSION_C` and to a lesser extent `KEXINIT_C`) needs the keys to be initialized. This is only the case after the exchange of `KEXINIT` messages. So all `NEWKEYS` and `KEXDH_INIT` messages sent before this happens will terminate the program.
- In `COMMUNICATION` the dispatch table is set in `serverloop.c` in such a way that it only recognizes the `KEXINIT` message and not the others.

5. Evaluation

5.1. Evaluation of OpenSSH

The intention is not to evaluate the entirety of OpenSSH based on a fairly limited test. Limited in the sense that we did not test every component of SSH. What we can do is conclude on the basis of the results from the last section that OpenSSH implements the Transport Layer fairly accurately. Again, the parallelism is missing for the most part but that was expected and already known from [8]. In a sense, the testing corroborates this. When we removed the parallelism from the model, it fit like a glove.

The question of whether we should allow this difference with the specification remains. As stated earlier, this difference is partially due to limitations on the testing techniques used. If both parties send their `KEXINIT` packets first and then read those packets, it will look like there is an order. In fact, for the test application it looks as though the client always sends its `KEXINIT` first, and then server sends its own packet while a code review clearly shows that both parties send their `KEXINIT` packets first. The same holds for the `NEWKEYS` and `KEXDH_INIT` type packets. It think it reasonable that to conclude that these discrepancies with the benign SSH server model are due to a limitation on the testing techniques due the limited ability to model parallelism in labeled DFA's. These transgressions, if they even can be called that, should be allowed.

The fact that the `VERSION` order is fixed *is* a problem. `VERSION_S` is always before `VERSION_C`. The RFC's don't specify an order to these messages, but OpenSSH has chosen such an order. This will cause problems when an OpenSSH implementation tries to connect to some other implementation that chooses differently. Both parties will end up waiting for the other party to start sending a version string. Note that if an implementation implements the parallelism correctly, it will be able to connect to an OpenSSH implementation without problems. This particular transgression could be

problematic. However, it seems that the most common implementations of SSH make the same choice in the matter and so no problems occur in practical usage.

As for the unexpected messages, the OpenSSH server has succeeded in handling every packet from the transport layer the test driver could throw at it. Of course, in this case handling means terminating the program. It is not clear from the RFC's if this is always the correct behaviour, they have some notion of sending `SSH_MSG_UNIMPLEMENTED` but they do not state which messages are considered unimplemented[8]. Terminating the process seems like a safe bet in any case.

5.2. Evaluation of Model-based Testing

Overall, using model-based testing techniques worked out rather well in the end. SSH is in principal suited to this kind of testing since there is quite a well defined notion of input, output and how these correlate. The way OpenSSH implemented however, negated this advantage quite expertly. That doesn't mean OpenSSH is poorly implemented as it does what it is supposed nicely as evidenced by the last section. Typical adapters for test drivers such as JTorX are easily implemented when used with code that is, for lack of a better term, functional. Functional in the sense that it does not manipulate state all the time. Alas, in OpenSSH pretty much every useful functions manipulates state or has a dependency on the state being a in certain condition. This does not make it impossible to write such an adapter, as evidenced by this thesis, but it does make it harder. An implementation of SSH could be created that deals with most of these issues. However, as far as I've been able to determine, this does not exist. Also, I feel as though any implementation will have some issues with state due to the nature of SSH. The effects could be lessened though by perhaps removing or limiting the use of the dispatch table.

In general the techniques applied are quite suited to a more complex system such as SSH if certain conditions are met. The solution presented in this thesis depends on the modifiability of the client and this is certainly not true for every system. Also, a model still has to be provided or created and the existence of a proper specification, let alone a decent model is not something that one always has for a system. These problems can be (partially) averted by means of model inference, the automatic learning of models from observable behaviour, as done in [5] with a tool such as the one in [9].

6. Conclusions

The primary goal of this thesis was to verify OpenSSH's implementation of the SSH Transport Layer Protocol by means of model-based testing techniques. As shown earlier, barring the issues with parallelism which are not really issues at all and were known beforehand, OpenSSH implements the specification almost exactly correct. As stated, the problems that were found were mostly due to a lack of proper modelling of parallelism leading to states and transitions that can not be observed. The problem concerning version exchange should be addressed. This could be done by either updating the specification to deal with the ambiguity concerning message order, or changing the implementation to support the parallelism found in the model.

Also, as a case study for model-based testing it was useful. Model-based testing is suitable for undertaking such a task as verifying a fairly complex system. The problems that arose during the creation of the adapter were problems that are likely to occur in quite a few systems. Hopefully, the solutions found in this thesis may help in solving those problems for other systems. However, the adapter may still be very difficult to implement even with a general idea on how to solve the problems that occur. State, especially when there is a lot of it , really provides numerous issues.

Another point is that the distinction between adapter and SUT can be blurry in systems such as OpenSSH where they are very similar and share much of the same code. For instance, do you use the client to test the server and vice versa? If so, what model do you use? The model of the client or the model of the server? It really is important to make this clear.

For future work, several things come to mind. First, and most importantly, testing the rest of the implementation using similar methods. In order to do that a full formal model of the SSH protocols and components is necessary. As for model inference, this could be used to ease the creation of said formal models and to check whether the work contained in this thesis was correct. It won't prove it correct, but it will provide an argument for its (in)correctness.

7. Acknowledgements

I would like use this section for expressing my gratitude towards *a*) Erik Poll, for supervising and for providing the initial idea for this thesis; *b*) all the people, be they friends, family or strangers that have aided me in other ways during the process of writing this thesis. And of course for tolerating my rants about some part of the OpenSSH source that eluded me[2].

References

[1] JTorX. `http://https://fmt.ewi.utwente.nl/redmine/projects/jtorx`.

[2] OpenSSH. `http://www.openssh.com/`.

[3] The OpenBSD project. `http://www.openbsd.org/`.

[4] yEd. `http://www.yworks.com/`.

[5] F. Aarts, E. Poll, and J. de Ruiter. Formal models of banking cards for free. Unpublished. 2012.

[6] L. Frantzen and J. Tretmans. Model-based testing of environmental conformance of components. In *Proceedings of the 5th international conference on Formal methods for components and objects*, pages 1–25. Springer-Verlag, 2006.

[2] This typically meant a segmentation fault.

[7] E. Poll and A. Schubert. Verifying an implementation of SSH. In *WITS*, volume 7, pages 164–177, 2007.

[8] E. Poll and A. Schubert. Rigorous specifications of the SSH Transport Layer. *Technical Report ICIS-R11004, Radboud University Nijmegen*, 2011.

[9] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(5):393–407, 2009.

[10] Ylönen, T. RFC 4250, The Secure Shell (SSH) Protocol Assigned Numbers. `http://tools.ietf.org/html/rfc4250`, 2006.

[11] Ylönen, T. RFC 4251, The Secure Shell (SSH) Protocol Architecture. `http://tools.ietf.org/html/rfc4251`, 2006.

[12] Ylönen, T. RFC 4252, The Secure Shell (SSH) Authentication Protocol. `http://tools.ietf.org/html/rfc4252`, 2006.

[13] Ylönen, T. RFC 4253, The Secure Shell (SSH) Transport Layer Protocol. `http://tools.ietf.org/html/rfc4253`, 2006.

[14] Ylönen, T. RFC 4254, The Secure Shell (SSH) Connection Protocol. `http://tools.ietf.org/html/rfc4254`, 2006.

A. middleman.pl

```
#! /usr/bin/env perl
#
#   middleman.pl
#   A middleman (hence the name) between the test driver and the edited openssh.
#
#   Copyright (c) 2012 Erik Boss <erik@erikboss.nl>
#

use warnings;
use strict;

use v5.10;

use IO::Pty::Easy;
use IO::Select;
use Getopt::Long qw/:config bundling/;

$| = 1;

my $verbose = 0; # Verbosity of the SSH process
my $quiet = 0; # Turn off debug output of the adapter
GetOptions(
    'verbose|v+' => \$verbose,
    'quiet|q' =>\$quiet,
);


# Messages I accept.
my
@messages = qw/
    !CONNECT
    !VERSION_C
    !VERSION_S
    !KEXINIT_C
    !KEXINIT_S
    !KEXDH_INIT
    !KEXDH_REPLY
    !NEWKEYS_C
    !NEWKEYS_S
    !IGNORE
    !DISCONNECT
    !UNIMPLEMENTED
    !ERROR/;

# (Hardcoded) path to program and its arguments.
my @args;
push(@args, "/home/erik/projects/current/bachelor-thesis/src/openssh-6.0p1/ssh");
if ($verbose > 0) {
    push(@args, '-' . ('v' x $verbose));
}
push(@args, "sshtest");
```

```
# Initialize the pseudo tty.
my $pty = IO::Pty::Easy->new;

# Spawn SSH
syswrite(STDERR, "Spawning SSH client... ") if not $quiet;
$pty->spawn(@args);

if (!$pty->is_active()) {
    if (not $quiet) {
        die "FAILED\n";
    } else {
        die;
    }
}
else {
    syswrite(STDERR, "SUCCESS\n") if not $quiet;
}

# Initialize the set of input handles.
my $read_set = new IO::Select();
$read_set->add($pty);
$read_set->add(\*STDIN);

while ($pty->is_active()) {
    # Get all active/open input handles.
    my ($rh_set) = IO::Select->select($read_set, undef, undef, undef,0);

    for my $rh (@$rh_set) {
        my ($res, $tmp) = ("","");

        # Is there is input waiting in the openssh process?
        if ($rh == $pty) {
            # Read all the input waiting.
            $res = $pty->read(1);

            # Split what was read into separate messages.
            my @results = split("\n",$res);
            for my $res (@results) {
                chomp($res);
                syswrite(STDERR, "SSH: --$res--\n") if not $quiet;;
                # Is the message of known type?
                if ($res ~~ @messages) {
                    # Print the message (to the test driver).
                    syswrite(STDOUT, "$res\n");
                    syswrite(STDERR, "Wrote '$res' to STDOUT.\n") if not $quiet;
                }
            }
        }
        elsif ($rh == \*STDIN) {
            # Read the message up to (and including) the newline.
            my $status;
            do {
                $status = sysread($rh, $tmp, 1);
```

```
            $res .= $tmp;
        }
        while($status > 0 && !($tmp eq "\n"));

        # Write message to the openssh process.
        $pty->write($res,0);

        chomp($res);
        syswrite(STDERR, "Wrote $res to Pty.\n") if not $quiet;
    }
    else {
        die "This is not supposed to happen...\n";
    }
  }
}
# Cleanup
$pty->close();

1;
```

B. adapterutil.c

```
#include "adapterutil.h"

#include "includes.h"

#ifdef USE_ADAPTER

#include <string.h>
#include <unistd.h>

#include "atomicio.h"
#include "kex.h"
#include "packet.h"
#include "roaming.h"
#include "ssh1.h"
#include "ssh2.h"

Kex *kex;

adapter_state *state = NULL;

void adapter_setup() {
    state = malloc(sizeof(adapter_state));
    state->busy = 0;
}

void register_kex (Kex *kex_) {
    kex = kex_;
    if (ADAPTER_DEBUG)
        fprintf (stderr,"kex registered ...\n");
}
```

```
int adapter_is_busy () {
    return state->busy;
}

int send_adapter_packet(int type, int read) {
    if (ADAPTER_DEBUG)
        fprintf (stderr, "Attempting to send '%s'\n", type_str (0,1,type));
    if (type == SSH2_MSG_KEXINIT || type == SSH2_MSG_KEXDH_INIT ||
        type == SSH2_MSG_NEWKEYS || type == SSH2_MSG_DISCONNECT ||
        type == SSH2_MSG_IGNORE || type == ADAPTER_MSG_VERSION) {

        if (state->busy == 1) {
            if (ADAPTER_DEBUG)
                fprintf (stderr, "Packet comes from adapter, letting it through...\n");
            return 0;
        }
        char *message_from_driver;
        int message;
        if (read == 1) {
            message_from_driver = read_message_from_driver();
            message = from_type_str(message_from_driver);
        }
        else
            message = type;

        if (ADAPTER_DEBUG)
            fprintf (stderr, "message: %s(%d), type: %s(%d), read: %d\n",
                    type_str (0,1,message), message, type_str (0,1,type), type, read);
        int connection_out;
        char *buf;

        if (read == 0 || message != type) {
            state->busy = 1;
            switch(message) {
                case SSH2_MSG_KEXINIT:
                    // As seen in clientloop.c
                    kex->done = 0;
                    kex_send_kexinit (kex);
                    packet_write_wait ();
                    break;
                case SSH2_MSG_KEXDH_INIT:
                    kexecdh_client (kex);
                    break;
                case SSH2_MSG_NEWKEYS:
                    kex_finish (kex);
                    break;
                case SSH2_MSG_DISCONNECT:
                    packet_disconnect("Disconnected due to adapter");
                    break;
                case SSH2_MSG_IGNORE:
                    packet_send_ignore(8);
                    break;
                case ADAPTER_MSG_VERSION:
                    connection_out = packet_get_connection_out();
```

```
                    buf = "SSH-2.0-OpenSSH_6.0\r\n"; // Hardcoded client version string.
                    atomicio(vwrite, connection_out, buf, strlen(buf));
                    break;
                default:
                    /* respond_to_driver("!ERROR");*/
                    if (ADAPTER_DEBUG)
                        fprintf(stderr,"Wait wut?\n");
                    state->busy = 0;
                    return 1;
            }
            if (state->busy == 1) {
                state->busy = 0;
                return 1;
            }
        }
    }
    else {
        if (ADAPTER_DEBUG)
            fprintf(stderr, "Packet is not part of the transport layer, letting it through...\n");
    }
    return 0;
}

char * read_message_from_driver() {
    size_t message_size;
    char *message;

    message_size = ADAPTER_MSG_SIZE;
    message = (char *) malloc(message_size + 1);
    int chars_read = getline(&message, &message_size, stdin);

    if (chars_read > 0)
        message[chars_read - 1] = '\0';

    return message;
}

void respond_to_driver(char *response) {
    if (strcmp(response,"") != 0) {
        fprintf(stdout, "%s\n", response);
        fflush(stdout);
    }
}

char *type_str(int output, int client, int type) {
    char *message, *prefix, *suffix;

    message = (char *) malloc(ADAPTER_MSG_SIZE);

    switch(output) {
        case 0 : prefix = "?"; break;
        case 1 : prefix = "!"; break;
        default : prefix = ""; break;
    }
```

```
    switch( client ) {
        case 0 :  suffix  = "_S"; break;
        case 1 :  suffix  = "_C"; break;
        default :  suffix  = ""; break;
    }

    switch(type) {
        case ADAPTER_MSG_VERSION: message = "VERSION"; break;
        case SSH2_MSG_KEXINIT : message = "KEXINIT"; break;
        case SSH2_MSG_NEWKEYS : message = "NEWKEYS"; break;
        case SSH2_MSG_KEXDH_INIT :message = "KEXDH_INIT"; suffix = ""; break;
        case SSH2_MSG_KEXDH_REPLY : message = "KEXDH_REPLY"; suffix = ""; break;
        case SSH2_MSG_CHANNEL_OPEN :
        case SSH2_MSG_CHANNEL_OPEN_CONFIRMATION :
        case SSH2_MSG_CHANNEL_EOF :
        case SSH2_MSG_CHANNEL_CLOSE :
        case SSH2_MSG_CHANNEL_REQUEST :
        case SSH2_MSG_CHANNEL_SUCCESS : message = "CHANNEL_TRAFFIC"; break;
        case SSH2_MSG_CHANNEL_DATA : message = "TRAFFIC"; break;
        case SSH2_MSG_UNIMPLEMENTED : message = "UNIMPLEMENTED"; suffix = ""; break;
        case SSH2_MSG_IGNORE : message = "IGNORE"; suffix = ""; break;
        case SSH2_MSG_DISCONNECT : message = "DISCONNECT"; suffix = ""; break;
        case SSH2_MSG_CHANNEL_FAILURE : message = "ERROR"; break;
        default :  sprintf (message, "UNKNOWN(%d)", type);
            break;
    }

    char * result ;
     result  =  (char *) malloc(ADAPTER_MSG_SIZE);
     sprintf ( result , "%s%s%s", prefix, message,  suffix );
    return  result ;
}

int  from_type_str (char * type_str ) {
    if  ( type_str  != NULL && strlen(type_str) > 4) {
        if  ( type_str [ strlen ( type_str ) - 1] == '\n')
            type_str [ strlen ( type_str ) - 1] = '\0';

        if  (strcmp( type_str +1, "IGNORE") == 0)
            return  SSH2_MSG_IGNORE;
        if  (strcmp( type_str +1, "UNIMPLEMENTED") == 0)
            return  SSH2_MSG_UNIMPLEMENTED;
        if  (strcmp( type_str +1, "DISCONNECT") == 0)
            return  SSH2_MSG_DISCONNECT;
        if  (strcmp( type_str +1, "KEXDH_INIT") == 0)
            return  SSH2_MSG_KEXDH_INIT;
        if  (strcmp( type_str +1, "KEXDH_REPLY") == 0)
            return  SSH2_MSG_KEXDH_REPLY;

        char *msg = (char *) malloc(256);
        strcpy (msg,  type_str +1);
        msg[strlen (msg) - 2] = '\0';
        if  (strcmp(msg, "KEXINIT") == 0)
            return  SSH2_MSG_KEXINIT;
```

```
        if (strcmp(msg, "NEWKEYS") == 0)
            return SSH2_MSG_NEWKEYS;
        if (strcmp(msg, "VERSION") == 0)
            return ADAPTER_MSG_VERSION;
    }
    return -1;
}

#endif
```

C. sshconnect.c

```
int
ssh_connect(const char *host, struct sockaddr_storage * hostaddr,
    u_short port, int family, int connection_attempts, int *timeout_ms,
    int want_keepalive, int needpriv, const char *proxy_command)
{
    // :
    // OpenSSH code for setting up the connection.
    // ;
#ifdef USE_ADAPTER
    adapter_setup();
    respond_to_driver("!CONNECT");
#endif
    // :
    // More OpenSSH code for finalizing the connection.
    // :
}

void
 ssh_exchange_identification (int timeout_ms)
{
    // :
    // OpenSSH code for getting the version number from the server.
    // :
#ifdef USE_ADAPTER
    respond_to_driver("!VERSION_S");
#endif
    // :
    // OpenSSH code for checking whether the versions match.
    // :
#ifdef USE_ADAPTER
    if (send_adapter_packet(ADAPTER_MSG_VERSION, 1) == 1)
        return;
#endif
    // :
    // OpenSSH code for sending the client version string to the server.
    // :
}
```

D. packet.c

```
int
 packet_read_poll_seqnr ( u_int32_t  *seqnr_p)
{
     u_int  reason, seqnr;
     u_char  type;
     char *msg;

     for  (;;)  {
          if  (compat20) {
               type  =  packet_read_poll2(seqnr_p);
               if  (type)  {
                    active_state ->keep_alive_timeouts = 0;
                   DBG(debug("received packet type %d", type));
               }
#ifdef USE_ADAPTER
               if  (ADAPTER_DEBUG)
                    fprintf ( stderr ,  "Read: %s(%d)\n", type_str(1,0,type),type);
              respond_to_driver ( type_str  (1,0, type));
#endif
     // Original  OpenSSH code. Returns the type of the message read.
}

 static  void
packet_send2(void)
{
     struct  packet *p;
     u_char  type, *cp;

     cp  =  buffer_ptr(& active_state ->outgoing_packet);
     type  = cp[5];
#ifdef USE_ADAPTER
     int  read  = 1;
     if  (type  == SSH2_MSG_CHANNEL_DATA) {
          buffer_consume(& active_state ->outgoing_packet, 14);
          char *adapter_message  =  buffer_ptr(& active_state ->outgoing_packet);
          adapter_message[ buffer_len (& active_state ->outgoing_packet)  - 1] = '\0';
          type  =  from_type_str(adapter_message);
          read  = 0;
     }
     if  (send_adapter_packet(type,  read)  == 1)
          return;
#endif
     // Original  OpenSSH code.
}
```