Verifying an implementation of SSH

Erik Poll Aleksy Schubert

Security of Systems (SoS)
Radboud University Nijmegen,
and
Warsaw University

work supported by EU project Mobius
and EU Marie Fellowship Sojourn

Information Society
Technologies

Motivation

Case study in checking the security of a program

There is a lot of work on verifying security protocols,
but to secure the weakest link we should look

- *not* at the cryptographic primitives
- *not* at the security protocol
- but at the software implementing this

Context: EU project Mobius

- Certifying security of mobile code
  - formal guarantees about security properties of code
  - by means of Proof Carrying Code (PCC)
- Focus for case studies on J2ME CLDC applications
  - Java-enabled mobile phones
- Real interest from telco's in
  - checking security properties of code which go beyond what the Java sandbox can provide
  - more rigorous methods than testing as the basis for putting digital signatures of code

The MIDP-SSH application

- Open source SSH client for Java-enabled mobile phones
  - SSH is a protocol similar to SSL

  Provides a secure shell
  - ie. confidentiality & integrity of network traffic

- SSH (v2) is secure, but what about this implementation?

Our analysis proceeded in two stages

6. informal, ad-hoc code inspection
7. formal, systematic verification

1. Flaws found in ad-hoc, manual code inspection

- Weak/no authentication

  no storage of public server keys
  - but fingerprint (hash value) is reported
- Poor use of Java access control (ie. visibility modifiers)

  ```
  public static java.lang.Random rnd = ...;
  final static int[] blowfish_sbox = ...;
  ```
  - Such bugs can be pointed out by automated tools, eg. Findbugs, or prevented by tools, eg. JAMIT tool for automated tightening of acces modifiers
  - Not a real threat (yet) on MIDP, due to current limits on running multiple applications.
- Lack of input validation

  missing checks for terminal control characters

2. Formal, systematic inspection

- Code annotated with formal specification language JML
  - specifying pre/postconditions and invariants
- Annotations checked with ESC/Java2 tool
  - lightweight program verification aka extended static checking

Two steps in use of JML and ESC/Java2:

e) proving exception freeness
  - ie. absence of unexpected runtime exceptions

f) proving adherence to functional spec

2a. Proving exception freeness

Example JML annotations specifying preconditions needed to rule out Nullpointer & ArrayIndexOutOfBounds-exceptions

```
/*@ requires
  @      0 <= s && s <= foo.length &&
  @      0 <= l && l <= foo.length - s;
  @*/
public void update(/*@ non_null @*/ byte foo[],
                   int s, int l)
 {  ... }
```

ESC/Java2 will warn if method calls to **update** violate this precondition, at compile-time

2a. Proving exception freeness

Example JML annotations specifying object invariants needed to rule out ArrayIndexOutOfBoundExceptions

```
public class SshPacket2 extends SshPacket {
    ...
 private int position;
 /*@ invariant phase_packet == PHASE_packet_length ==>
   @         (0 <= position && position < packet_array.length);
   @*/
 /*@ invariant (phase_packet == PHASE_block && !finished) ==>
   @                (0 <= position && position < block.length);
   @*/
```

ESC/Java2 will warn if these invariants are violated, at compile-time

2a. Proving exception freeness

Results:

- Improvements in code needed to avoid some runtime exceptions
  - esp ArrayIndexOutOfBoundsExceptions, that could occur when handling of malformed packets

  Note that
    - such cases are hard to catch using testing, because of huge search space of possible malformed packets
    - in a C(++) application these bugs would be buffer overflow vulnerabilities!
- Also spotted: a missing check of a MAC (Message Authentication Code)
  - process of annotating code *forces* a thorough code inspection

Beyond proving exception freeness: proving functional correctness

- Exception freeness looks at what application should *not* do
  - it should not crash with unexpected runtime exceptions
- How about looking at what it *should* do ?

- This requires some formal specification of the SSH protocol

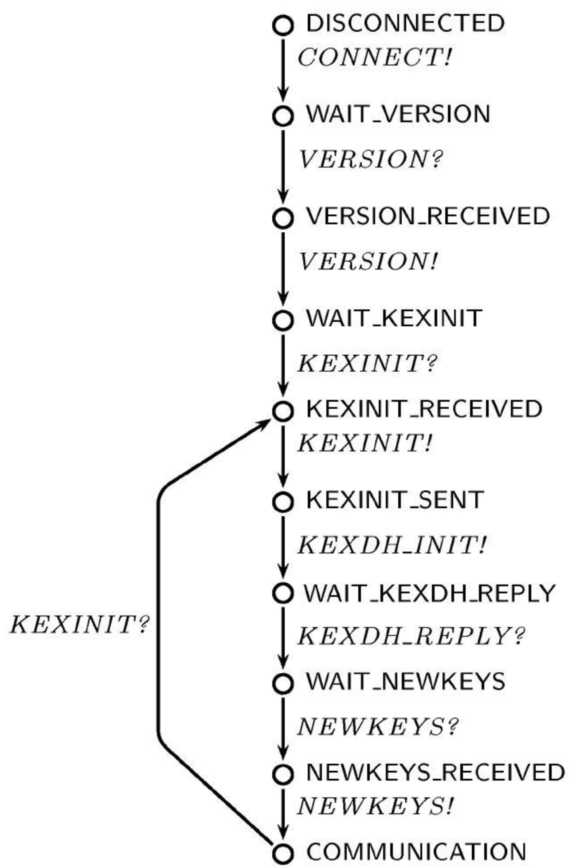
The SSH protocol

- Official specification given in RFCs 4250-4254
  - Over 100 pages of text
  - Many options & variants
    - effectively, SSH is a collection of protocols

- The official specification far removed from typical formal description of security protocols.

- We defined a partial formal specification of SSH as Finite State Machine (FSM) aka automaton
  - SSH client effectively implements a FSM, which has to respond to 20 kinds of messages in right way

The basic SSH protocol as FSM



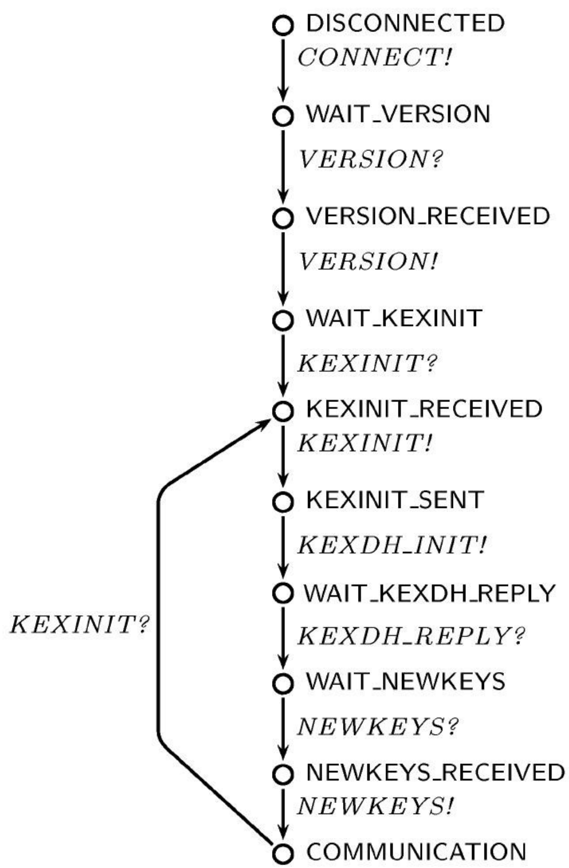
This FSM defines a typical, correct protocol run

SSH as abstract security protocol

- This FSM can also be written in the common notation used for security protocol verification

1. $C \rightarrow S$: CONNECT
2. $S \rightarrow C$: V_C // VERSION of the server
3. $C \rightarrow S$: V_S // VERSION of the client
4. $S \rightarrow C$: I_S // KEXINIT
5. $C \rightarrow S$: I_C // KEXINIT
6. $C \rightarrow S$: exp(g,X) // KEXDH INIT
7. $S \rightarrow C$: K_S.exp(g, Y).$\{H\}_{inv(K_S)}$ // KEXDH REPLY
8. ...

The basic SSH protocol as FSM



However, this FSM defines

- only one correct protocol run
- no *in*correct protocol runs

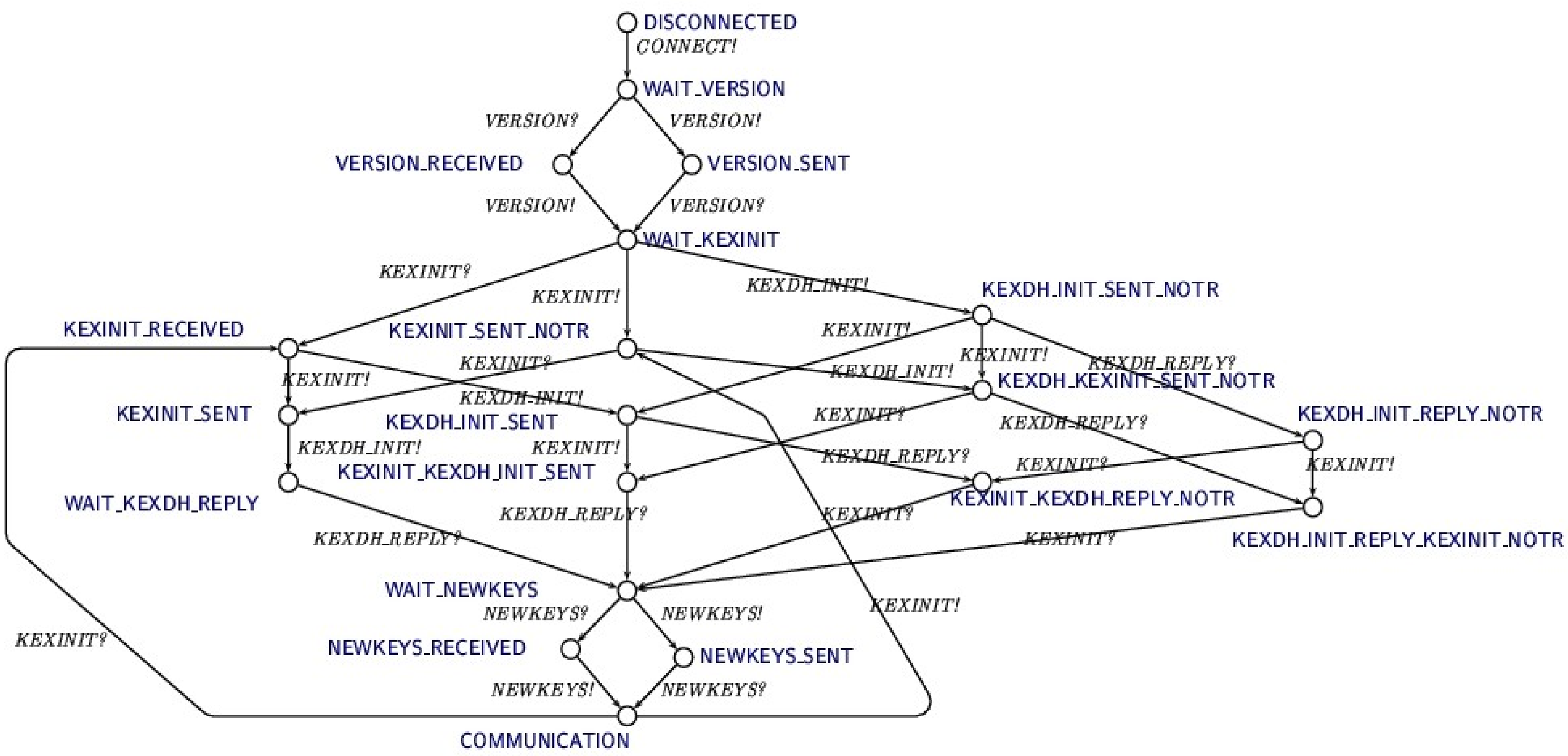
How do we specify:

vi. optional features in the RFCs, which allow various correct protocol runs?

vii. how *in*correct protocol runs should be handled?

Specifying SSH protocol as FSM (i)

Incl. optional features allowed by RFCs we get

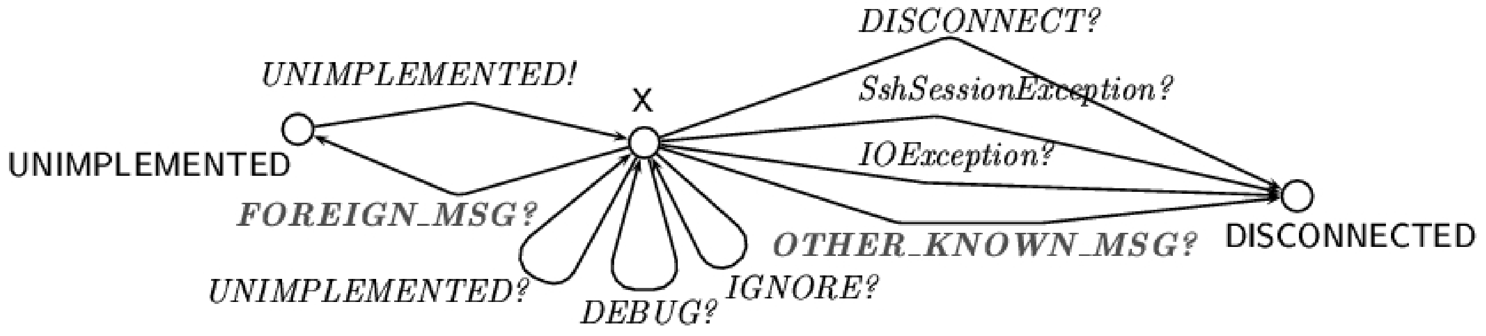


Specifying SSH protocol as FSM (ii)

To handle incorrect runs, there are, in every state X, additional messages that

- should be ignored, *or*
- should be ignored after a reply "UNIMPLEMENTED", *or*
- should lead to disconnection.

In every state X, we have to add an 'aspect' of the form below



Specifying SSH protocol as FSM

- Obtaining these FSM from the informal specification of SSH given in the RFCs is hard:
  - notion of state is completely implicit in the RFCs
  - constraints of correct sequences of messages given in many places
    - Eg constraints such as "once a party sends a SSH_MSG_KEXINIT message [. . .], until it sends a SSH_MSG_NEWKEYS message, it MUST NOT send any messages other than [. . .]"
  - not clear if underspecification is always deliberate
    - eg order of VERSION messages from client to server and vv.

- Note that anyone implementing SSH will effectively have to extract the same information from the RFCs as is given by our FSM

2b. Verifying the code against FSM

- AutoJML tool used to produce JML annotations from FSM
  - tool extended to cope with multiple of diagrams
- Obvious security flaw:

  implementation doesn't record the state correctly (at all!)
  - Hence, an attacker can ask for username/password before session key has been established
- Improved code was successfully verified against the FSM

Effort

- Formal specification & verification of the protocol implementation (4.5 kloc) took around 6 weeks
  - ie. proving
    a) exception freeness, and
    b) adherence to our formal specification given by FSM

    a) catches errors in handling malformed messages
    b) catches errors in handling incorrect/unusual sequences of messages
  - incl. 2 weeks understanding & formalising SSH specs

Central problem: how to relate

typical abstract security protols:
tens of lines

?

official spec of SSH:
>100 pages of RFCs

code:
4.5 kloc of Java

How to formally specify SSH?

- Traditional format for specifying security protocols used for protocol verification

  Eg

  1. $C \to S$: CONNECT
  2. $S \to C : V_C$ // VERSION of the server
  3. $C \to S : V_S$ // VERSION of the client
  4. $S \to C : I_S$ // KEXINIT
  5. $C \to S : I_C$ // KEXINIT
  6. $C \to S : exp(g,X)$ // KEXDH INIT
  7. $S \to C : K_S.exp(g, Y).\{H\}_{inv(K_S)}$ // KEXDH REPLY

  cannot conveniently capture

  - options and allowed variants in the behaviour
  - required/allowed responses to deviations from this correct protocol run

How to formally specify SSH?

- Our FSM is an attempt to bridge the big gap between
  - real security protocols, and
  - formal descriptions of abstract protocols studied for protocol verification
- Bridging this gap could result in
  - better specs of real security protocols
  - formal verification of more realistic protocols

Conclusions – about MIDP-SSH

- Of course, an incorrect implementation of a secure protocol can be completely insecure...
- We successfully found & removed flaws from the MIDP-SSH implementation
  - by informal and formal methods

- Our verification can catch errors in handling
  - incorrectly formatted messages, and
  - incorrect sequences of messages
- But, our verification is not *complete*, as our formal specification is only a *partial* formal specification of SSH,

Conclusions – about SSH

- The official specification of SSH can be improved.

  In particular, including an explicit notion of state would help (and make security flaws as found in MIDP-SSH much less likely)

- Note that anyone implementing SSH will effectively have to extract the same information from the RFCs as is given by our FSM

Ongoing work

- FSM specification is still only a partial specification:
  - it specifies the order, but not format of messages

What would be a convenient format for a *complete* formal specification of SSH?

  - Graphical notation of FSM quickly becomes unwieldy

Future work

- Other implementations of SSH
- Other protocols , eg SSL/TLS
- Using FSM as basis for model-based testing to check for flaws in implementations

[For more info: http://www.cs.ru.nl/~erikpoll/papers/wits.pdf]