Model Learning and Model Checking of SSH Implementations
Introduction

• protocols: SSH, TLS, SMTP, FTP, TCP, UDP...
• many implementations per protocol
  ➢ implementations **MUST*/SHOULD/MAY adhere** to the specifications...
Introduction

• protocols: SSH, TLS, SMTP, FTP, TCP, UDP...
• many implementations per protocol
  ➢ implementations MUST/SHOULD/MAY adhere to the specifications...
Motivation

Model Learning

→ *automatically* infers models for concrete implementations
→ checking conformance of models may be difficult
Motivation

Model Learning

\(\rightarrow automatically\) infers models for concrete implementations

\(\rightarrow\) checking conformance of models may be difficult
Motivation

Model Checking
- *automatically* checks conformance of models to specifications
- *requires* models and formalized specifications

Model Learning
- *automatically* infers models for concrete implementations
- checking conformance of models may be difficult
Motivation

Model Learning + Model Checking

- automatically infers models for concrete implementations
- automatically checks conformance of models to specifications
- requires formalized specifications

Model Checking

- automatically checks conformance of models to specifications
- requires models and formalized specifications

Model Learning

- automatically infers models for concrete implementations
- checking conformance of models may be difficult
What was done

Model Learning + Model Checking
  \(\rightarrow\) \textit{automatically} infers models for concrete implementations
  \(\rightarrow\) \textit{automatically} checks conformance of models to specifications
  \(\rightarrow\) \text{\textit{requires}}\ formalized specifications

Application of ML+MC on SSH (a real world protocol):
  1. use Model Learning to infer models of 3 SSH server implementations
  2. formalize specifications from the SSH RFC standards
  3. use Model Checking to verify models against these specification
What was done

Model Learning + Model Checking

→ *automatically* infers models for concrete implementations
→ *automatically* checks conformance of models to specifications
→ *requires* formalized specifications

Schematic Overview

![Diagram showing Model Learning and Model Checking processes with spec formalization and SUL models.](image-url)
What was done

Model Learning + Model Checking

→ automatically infers models for concrete implementations
→ automatically checks conformance of models to specifications
→ requires formalized specifications

enough for a publication?
What was done

Model Learning + Model Checking

\(\rightarrow automatically\) infers models for concrete implementations

\(\rightarrow automatically\) checks conformance of models to specifications

\(\rightarrow requires\) formalized specifications

\underline{Model Learning:}\n
\(\rightarrow requires\) mapper component

\(\rightarrow requires\) thorough testing

\underline{Model Checking:}\n
\(\rightarrow requires\) model transformation

\(\rightarrow requires\) counterexample validation
Model Learning + Model Checking

→ *automatically* infers models for concrete implementations
→ *automatically* checks conformance of models to specifications
→ requires mapper component
→ requires formalized specifications
→ requires thorough testing
→ requires model transformation
→ requires counterexample validation

What was done

Patrick’s M. Thesis

Toon’s B. Thesis

Publication
What was done
Model Learning

Learner

SUL (System under Learning)

inputs
outputs

inferred state model
Model Learning

**Learner Queries:**
- register/ok

**Input Alphabet:**
- register, login, logout

**Output Alphabet:**
- ok, nok

Mealy Machine

**Input Alphabet:**
- register, login, logout

**Output Alphabet:**
- ok, nok
Model Learning

Learner Queries:
- register/ok
- login/nok
- logout/nok
- register/ok
- register/nok

Input Alphabet:
- [register, login, logout]
Model Learning

Learner Queries:
- register/ok
- login/nok
- logout/nok
- register register/ok nok

Hypothesis:
Model Learning

Hypothesis:
Model Learning

Test Queries:
register register login/ok nok nok

Hypothesis:
Model Learning

Hypothesis:
Model Learning

- Learner
  - New Hypothesis
  - New queries
  - counterexample
- SUL
- Tester
  - tests
  - correct!
Model Learning

Learner/Tester → SUL
Model Learning

Learner/Tester

SUL

Abstract i/o (strings)

- login, logout
- ok, nok

Concrete i/o

- SUL.login, SUL.logout
- true/false

- method calls, returned obj. packets
Model Learning

Learner/
Tester

SUL

login_0
login_1
...
ok, nok

small abstract
i/o alphabet

login(uid)
ok, nok

parameterized
i/o alphabet
Model Learning

abstract i/o  param i/o  concrete i/o

Learner/
Tester

Mapper

SUL

Mapper translates:
1. between abstract and param. i/o
2. between param. i/o and concrete i/o
Model Learning

Mapper
1. translates:
   - between abstract and param. i/o
   - between param. i/o and concrete i/o
2. gives a (deterministic) Mealy Machine representation
   - removes time dependencies, non-determinism.
Model Learning

Learner -> Mapper -> SUL
queries -> concrete queries/tests
New Hypothesis -> counterexample
Tester -> correct!

queries

correct!
Model Learning

Learner

Mapper

SUL

New Hypothesis

queries

concrete queries/tests

correct!

counterexample

tests

prop1_LTL

prop2_LTL

SUL:

prop1_LTL

prop2_LTL

Model Checking

Spec. formalization

RFC

Model Learning Spec.
Model Learning

TODOs:
1. know your SUL
2. define i/o alphabet
3. implement mapper
4. choose learner and tester algorithms
5. connect and execute!
Model Learning

**TODOs:**
1. know your SUL
2. define i/o alphabet
3. implement mapper
4. choose learner and tester algorithms
5. connect and execute!
The SSH Protocol

- protocol for operating network services (e.g. terminal) securely over an unsecured network
- client/server application layer protocol, runs on top of TCP
The SSH Protocol

- protocol for operating network services (e.g. terminal) securely over an unsecured network
- client/server application layer protocol, runs on top of TCP
- Learner + Mapper replaces the SSH CLIENT, goal learn the SSH Server!
The SSH Protocol

➢ comprises three layers which *interoperate (no encapsulation)*
➢ each layer responsible for each of the 3 protocol steps,
➢ for each we define the *happy flow* at an abstract level

<table>
<thead>
<tr>
<th>User Authentication Layer</th>
<th>Connection Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP Layer</td>
<td></td>
</tr>
<tr>
<td>Transport Layer</td>
<td></td>
</tr>
</tbody>
</table>

CLIENT APPLICATION

UNSECURE NETWORK

SERVER APPLICATION
The SSH Protocol

➢ 3 steps
  1. establish a secure connection (by exchanging keys)
The SSH Protocol

➢ 3 steps
1. establish a secure connection (by exchanging keys)
The SSH Protocol

➢ 3 steps
  1. establish a secure connection (by exchanging keys)
     1. exchange preferences (KEXINIT)
     2. perform key exchange (KEXxx)
     3. put new keys to use (NEWKEYS)
     4. engage the auth. service (SR_AUTH)

Happy flow:

Other inputs: DEBUG, IGNORE, DISCONNECT.
Other outputs: DEBUG, IGNORE, DISCONNECT.
The SSH Protocol

➢ 3 steps
  1. establish a secure connection (by exchanging keys)
The SSH Protocol

- 3 steps
  1. Establish a secure connection (by exchanging keys)
  2. Key re-exchange (rekey): same procedure, old keys are replaced by new ones

- Can happen any time after the initial key exchange protocol should not affect operation of higher layer protocols

User Authentication Layer | Connection Layer
---|---
Transport Layer
TCP/IP Layer

CLIENT APPLICATION
UNSECURE NETWORK
SERVER APPLICATION
The SSH Protocol

3 steps
1. establish a secure connection (by exchanging keys)
2. authentication with server

User Authentication Layer
Connection Layer
Transport Layer
TCP/IP Layer
The SSH Protocol

➢ 3 steps
1. establish a secure connection (by exchanging keys)
2. authentication with server
   ▪ user/public key auth. UA_PK_OK
   ▪ user/password auth. UA_PW_OK
   ▪ none auth. UA_NONE

Happy flow:

Other inputs: UA_NONE, UA_PK_NOK, UA_PW_NOK...
Other outputs: UA_FAILURE
The SSH Protocol

➢ 3 steps
1. establish a secure connection (by exchanging keys)
2. authentication with server
The SSH Protocol

➢ 3 steps
1. establish a secure connection (by exchanging keys)
2. authentication with server
3. access network services (say remote terminal)
Learning SSH

➢ 3 steps

1. establish a secure connection (by exchanging keys)
2. authentication with server
3. access network services (say remote terminal)

1) open channel (CH_OPEN)
2) request term. service over channel (CH_REQUEST_PTY)
3) channel data management (CH_SEND_DATA)
4) close channel (CH_CLOSE)
Learning SSH

TODOs:
1. know your SUL
2. define i/o alphabet
3. implement mapper
4. choose learner and tester algorithms
5. connect and execute!

Mapper task
1. *translate between abstract, parametrized and concrete i/o*

AUTH_PW_OK

AUTH_REQUEST(“password”, “john”...)
Learning SSH

1. **Translator**
   - translates between abstract, parametrized and concrete i/o
   - needs to be able to encrypt/decrypt, compress/decompress
   - stores information in variables: encryption keys, session ID, sequence number...
   - implemented by adapting an existing SSH suite implementation (Paramiko)

**AUTH_PW_OK**  **AUTH_REQUEST(“password”, “john”...)**
Learning SSH

TODOs:
1. know your SUL
2. define i/o alphabet
3. implement mapper
4. choose learner and tester algorithms
5. connect and execute!

Mapper task
1. translate between abstract, parametrized and concrete i/o
   ➢ needs to be able to encrypt/decrypt compress/decompress
   ➢ stores information in variables: encryption keys, session ID, sequence number...
     ➢ implemented by adapting an existing SSH suite implementation (Paramiko)

2. ensure deterministic Mealy Machine representation
   ➢ reliable setting of timing parameters (e.g. NO_RESP timing parameter)

false NO_RESP, mapper should have waited longer
Learning SSH

TODOs:
1. know your SUL
2. define i/o alphabet
3. implement mapper
4. choose learner and tester algorithms
5. connect and execute!

Mapper task
1. translate between abstract, parametrized and concrete i/o
   - needs to be able to encrypt/decrypt compress/decompress
   - stores information in variables: encryption keys, session ID, sequence number...
     ➢ implemented by adapting an existing SSH suite implementation
     (Paramiko)

2. ensure deterministic Mealy Machine representation
   - reliable setting of timing parameters (e.g. NO_RESP timing parameter)
   - enforce one output per input by concatenating (‘_’) multiple responses to an input into one output
Learning SSH

Mapper task
1. **translate between abstract, parametrized and concrete i/o**
   - needs to be able to encrypt/decrypt compress/decompress
   - stores information in variables: encryption keys, session ID, sequence number...
     ➔ implemented by adapting an existing SSH suite implementation
     (Paramiko)

2. **ensure deterministic Mealy Machine representation**
   - reliable setting of timing parameters (e.g. NO_RESP timing parameter)
   - enforce one output per input by concatenating (’_’) multiple responses to an input into one output

TODOs:
1. know your SUL
2. define i/o alphabet
3. **implement mapper**
4. choose learner and tester algorithms
5. connect and execute!
Learning SSH

**LearnLib algorithms:**
- L*
- Observation Pack

**Tester Algorithms:**
- Random Walk
- W Method
- Yannakakis (Random + Exhaustive)

**TODOs:**
1. know your SUL
2. define i/o alphabet
3. implement mapper
4. **choose learner and tester algorithms**
5. connect and execute!
Learning SSH

TODOs:
1. know your SUL
2. define i/o alphabet
3. implement mapper
4. **choose learner and tester algorithms**
5. connect and execute!

Note on testing:
- testing can never guarantee correctness
- exhaustive test algs. ensure a well defined level of confidence
  - *but lack penetration*
- random test algs. have penetration ➔ more likely to find CEs
  - *but give no formal confidence*
Learning SSH

Example Yannakakis

→ random:
  - choose bigger k
  - random mid sequences

→ exhaustive
  - choose smaller k
  - generate for all mid-sequences
  - attain confidence

TODOs:
1. know your SUL
2. define i/o alphabet
3. implement mapper
4. choose learner and tester algorithms
5. connect and execute!

We used:
Random Yannakakis (k=4)
Exhaustive Yannakakis (k=2)
Learning SSH

TODOs:
1. know your SUL
2. define i/o alphabet
3. implement mapper
4. choose learner and tester algorithms
5. connect and execute!

SSH Server Implementations

LearnLib: Observation Pack

Paramiko Adaptation

queries

counterexample

tests

Learner

Mapper

SUL

Yannakakis Tester

correct!

concrete queries/tests
Learning SSH Results

Open SSH Learned Model
Learning SSH Results

Open SSH Learned Model peculiarities:
- authentication
- rekey changes state
- max one terminal
## Learning SSH Results

<table>
<thead>
<tr>
<th>SUT</th>
<th>States</th>
<th>Hypotheses</th>
<th>Mem. Q.</th>
<th>Test Q.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSH 6.9p1-2</td>
<td>31</td>
<td>4</td>
<td>19836</td>
<td>76418</td>
</tr>
<tr>
<td>BitVise 7.23</td>
<td>65</td>
<td>15</td>
<td>24996</td>
<td>58423</td>
</tr>
<tr>
<td>DropBear v2014.65</td>
<td>29</td>
<td>8</td>
<td>8357</td>
<td>64478</td>
</tr>
</tbody>
</table>

- rekey (3 step sequence)
- buffering
- mapper induced behavior
Learning SSH Results

- rekey (3 step sequence)
- buffering
- mapper induced behavior

<table>
<thead>
<tr>
<th>SUT</th>
<th>States</th>
<th>Hypotheses</th>
<th>Mem. Q.</th>
<th>Test Q.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSH 6.9p1-2</td>
<td>31</td>
<td>4</td>
<td>19836</td>
<td>76418</td>
</tr>
<tr>
<td>BitVise 7.23</td>
<td>65</td>
<td>15</td>
<td>24996</td>
<td>58423</td>
</tr>
<tr>
<td>DropBear v2014.65</td>
<td>29</td>
<td>8</td>
<td>8357</td>
<td>64478</td>
</tr>
</tbody>
</table>

- state permitting rekey
- rekey state
Learning SSH Results

- rekey (3 step sequence)
- buffering
- mapper induced behavior
  - remember, we learn SUL + mapper, not SUL alone

<table>
<thead>
<tr>
<th>SUT</th>
<th>States</th>
<th>Hypotheses</th>
<th>Mem. Q.</th>
<th>Test Q.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSH 6.9p1-2</td>
<td>31</td>
<td>4</td>
<td>19836</td>
<td>76418</td>
</tr>
<tr>
<td>BitVise 7.23</td>
<td>65</td>
<td>15</td>
<td>24996</td>
<td>58423</td>
</tr>
<tr>
<td>DropBear v2014.65</td>
<td>29</td>
<td>8</td>
<td>8357</td>
<td>64478</td>
</tr>
</tbody>
</table>
What was done

- Model Learning
- Spec. formalization
- Model Checking

SUL:
- prop1_LTL
- prop2_LTL

SUL:
- prop1_LTL: ✔
- prop2_LTL: ✗
Model Checking

➢ we used NuSMV:
  ➢ supports LTL, CTL and Real Time CTL specifications
  ➢ requires conversion to a .SMV model
Model Checking

- we used NuSMV:
  - supports LTL, CTL and Real Time CTL specifications
  - requires conversion to a .SMV model
  - wrote script to automatically perform this conversion

Mealy Machine

- \( q_0 \) (start)
- \( q_1 \)
- MSG/NOK
- BEGIN/OK
- MSG/ACK

NuSMV Model

- \( \text{MODULE main} \)
- \( \text{VAR state : \{q0, q1\};} \)
- \( \text{inp : \{BEGIN, MSG\};} \)
- \( \text{out : \{OK, NOK, ACK\};} \)
- \( \text{ASSIGN} \)
- \( \text{init(state) := q0;} \)
- \( \text{next(state) := case} \)
  - \( \text{state = q0 & inp = BEGIN: q1;} \)
  - \( \text{state = q0 & inp = MSG: q0;} \)
  - \( \text{state = q1 & inp = BEGIN: q1;} \)
  - \( \text{state = q1 & inp = MSG: q1;} \)
  - \( \text{esac;} \)
- \( \text{out := case} \)
  - \( \text{state = q0 & inp = BEGIN: OK;} \)
  - \( \text{state = q0 & inp = MSG: NOK;} \)
  - \( \text{state = q1 & inp = BEGIN: OK;} \)
  - \( \text{state = q1 & inp = MSG: ACK;} \)
  - \( \text{esac;} \)

\( \rightarrow \) kripke structure with:
- state function (next)
- output function (out)
Model Checking

➢ we used NuSMV:
  ➢ supports LTL, CTL and Real Time CTL specifications
  ➢ requires conversion to a .SMV model

Mealy Machine

NuSMV Model

\[ G \text{ inp}=\text{BEGIN} \rightarrow \text{out}=\text{OK} \]

\[ G \text{ out}=\text{OK} \rightarrow \]

\[ X \text{ inp}=\text{MSG} \rightarrow \text{out}=\text{ACK} \]

\[ G \cup \text{ out}=\text{OK} \rightarrow \]

\[ X \text{ inp}=\text{MSG} \rightarrow \text{out}=\text{NOK} \]
Model Checking

➢ we used NuSMV:
  ➢ supports LTL, CTL and Real Time CTL specifications
  ➢ requires conversion to a .SMV model

Mealy Machine

\[
\begin{align*}
\text{G (inp=BEGIN -> out=OK) } & \quad \checkmark \\
\text{G (out=OK ->} & \quad \checkmark \\
\text{X (inp=MSG -> out=ACK) } & \quad \checkmark \\
\text{G U (out=OK ->} & \quad \times \\
\text{X (inp=MSG -> out=NOK))} &
\end{align*}
\]
Model Checking

- we used NuSMV:
  - supports LTL, CTL and Real Time CTL specifications
  - requires conversion to a .SMV model
- specification either holds or counterexample (CE) given
  - CE may
    - agree with the SUL \(\rightarrow\) non-conformance
    - disagree with the SUL \(\rightarrow\) a CE for the learner
- thus, all CEs must first be confirmed by running it on the system
  - integrated model checker into testing s.t. all CEs are confirmed
Formalizing SSH Specifications

- LTL formulas with both forward and past modalities
- checked on the mapper + SUL assembly (not only on the SUL itself), thus
  results not fully translatable
- 4 types:
  - basic properties: describe the SUL + mapper setup, all true
  - security properties: define the overriding goal of each layer
  - rekey properties: is rekey allowed (does it not disconnect)
    does rekey preserve state?
  - functional properties: are MUST/SHOULD statements met
Formalizing SSH Specifications

Only one SSH connection is made and once it is gone, it is gone.

- basic properties
- security properties
- rekey properties
- functional properties

\[ G(\text{out}=\text{NO CONN} \rightarrow G(\text{out}=\text{NO CONN} | \text{out}=\text{CH MAX} | \text{out}=\text{CH NONE})) \]

SUL no longer responds
Formalizing SSH Specifications

We consider an transport layer state machine secure if there is:

- no path from the initial state to the point where the authentication service is invoked without exchanging and employing cryptographic keys.

\[
G(\text{hasReqAuth} \rightarrow O((\text{inp}=\text{NEWKEYS} & \text{out}=\text{NO RESP}) \& O((\text{inp}=\text{KEX30} & \text{out}=\text{KEX31_NEWKEYS}) \& O(\text{out}=\text{KEXINIT}))))
\]
Formalizing SSH Specifications

SSH_MSG_CHANNEL_CLOSE
Upon receiving this message, a party **MUST** send back an
SSH_MSG_CHANNEL_CLOSE unless it has already sent this
message for the channel.
(RFC 4254, p 9)

\[ G \ ( \text{hasOpenedChannel} \rightarrow \left( ( \text{inp} = \text{CH CLOSE}) \rightarrow ( \text{out} = \text{CH CLOSE}) \right) ) \]

\[ W \ ( \text{connLost} \mid \text{kexStarted} \mid \text{out} = \text{CH CLOSE}) \]
Formalizing SSH Specifications

SSH_MSG_CHANNEL_CLOSE
Upon receiving this message, a party **MUST** send back an
SSH_MSG_CHANNEL_CLOSE unless it has already sent this
message for the channel.
(RFC 4254, p 9)

\[
G( hasOpenedChannel \rightarrow \left( \left( inp=CH\ \text{CLOSE} \right) \rightarrow \left( out=CH\ \text{CLOSE} \right) \right) )
\]

\[
W( connLost \mid kexStarted \mid out=CH\ \text{CLOSE} )
\]

→ in red, predicates not expressed in RFC statement, yet deducted from context
→ formalization forces **clarification**
Formalizing SSH Specifications

SSH_MSG_USERAUTH_SUCCESS MUST be sent only once.
(RFC 4252 p. 5)

\[ G ( \text{out}=UA\ SUCCESS \rightarrow X \ G \ \text{out} \neq UA\ SUCCESS) \]

SSH_MSG_USERAUTH_SUCCESS has been sent, any further authentication requests received after that SHOULD be silently ignored.
(RFC 4252 p. 5)

\[ G ( \text{out}=UA\ SUCCESS \rightarrow \ X \ ( ( \text{authReq} \rightarrow \text{out} = \text{NO RESP} ) \ W (\text{connLost} \ | \ kexStarted) )) \]
Formalizing SSH Specifications

key exchange does not affect the protocols that lie above the SSH transport layer.
(RFC 4253 p. 24)

➢ state based property:
  ➢ cannot be efficiently formulated by LTL
  ➢ checked using script

➢ basic properties
➢ security properties
➢ rekey properties
➢ functional properties
## Model Checking Results

<table>
<thead>
<tr>
<th>Property</th>
<th>Key word</th>
<th>OpenSSH</th>
<th>Bitvise</th>
<th>DropBear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>Trans.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Auth.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Rekey</td>
<td>Pre-auth.</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Auth.</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Funct.</td>
<td>Prop. 6</td>
<td>MUST</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Prop. 7</td>
<td>MUST</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Prop. 8</td>
<td>MUST</td>
<td>X*</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Prop. 9</td>
<td>MUST</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Prop. 10</td>
<td>MUST</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Prop. 11</td>
<td>SHOULD</td>
<td>X*</td>
<td>X*</td>
</tr>
<tr>
<td></td>
<td>Prop. 12</td>
<td>MUST</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **UA_SUCC** once
- **NO_RESP** after **UA_SUCC**
- *(X sends UNIMPL)*
- **CH_CLOSE** after **CH_CLOSE**
Conclusions and Future Work

Conformance checking of the SSH protocol, using model learning & model checking

- inferred models for 3 SSH server implementations
- run extensive testing on models
- formalized and checked models against security properties, as well as server RFC MUST/SHOULD requirements
- found inconsistencies with limited security impact

Future work:

- formalize mapper so it is clear what it does (and a concretization can be made)
- make mapper abstraction less impactful ⇒ reduce num. of mapper induced states
- learn SSH client, model check assembly client/server
- replace classical learner by a register automata learner, extract parameters and infer their related behavior