**Automated reverse engineering 1 - Active state machine learning**

Howar, Falk, Malte Isberner, and Bernhard Steffen. "Tutorial: Automata Learning in Practice." Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change. Springer Berlin Heidelberg, 2014. 499-513.

De Ruiter, Joeri, and Erik Poll. "Protocol state fuzzing of TLS implementations." USENIX Security. 2015.

Aarts, Fides, et al. "Improving active Mealy machine learning for protocol conformance testing." Machine learning 96.1-2 (2014): 189-224.

Bauer, Oliver, Maren Geske, and Malte Isberner. "Analyzing program behavior through active automata learning." International Journal on Software Tools for Technology Transfer 16.5 (2014): 531-542.

Cho, Chia Yuan, Eui Chul Richard Shin, and Dawn Song. "Inference and analysis of formal models of botnet command and control protocols." Proceedings of the 17th ACM conference on Computer and communications security. ACM, 2010.

**Automated reverse engineering 2 - Passive state machine learning**

Marijn J.H. Heule and Sicco Verwer. 2013. Software model synthesis using satisfiability solvers. Empir. Softw. Eng. 18, 4 (August 2013), 825–856.

Walkinshaw, Neil, Russell Taylor, and John Derrick. "Inferring extended finite state machine models from software executions." Reverse Engineering (WCRE), 2013 20th Working Conference on. IEEE, 2013.

Comparetti, Paolo Milani, et al. "Prospex: Protocol specification extraction." Security and Privacy, 2009 30th IEEE Symposium on. IEEE, 2009.

Cui, Weidong, Jayanthkumar Kannan, and Helen J. Wang. "Discoverer: Automatic Protocol Reverse Engineering from Network Traces." USENIX Security. 2007.

**28-3 [STRE-16 Lecture 2]**

Computational complexity notions: NP, NP-hard, NP-complete and Undecidable

Learning theory: Field that studies learning problems, or how to find a model for data

Learning algorithm is an algorithm that given a dataset, produces a model. And then there are complexities involved: how difficult is it to find the smallest possible model and can we find the model that is provably close to the one that generates the data.

**Learning in the limit** [general for any machine learning algorithm] -> Studies learning from infinite data

Views learning as an ongoing process

1. Learner receives some data from a target concept C
2. Updates its hypothesis H, then goes to 1

Is only successful if it converges:

* L(H) = L(C) at some point in time t
* And L(H) = L(C) at all time points > t
* Learner does not need to know when it converges.

Learning DFAs from labeled data

Input: labeled data, positive and negative traces

Goal: Find the smallest DFA that is consistent with the data

Occam’s razor: among competing hypothesis, the one with the fewest assumptions should be selected.

Theorem: DFAs cannot be learned in the limit from unlabeled data!

Limit learners should at some point coverge to C

{a,b}\* contains all finite languages over {a,b}

Suppose a learner converges to {a,b}\* at some point.

At that point, the learner has seen a finite data set D

D is a finite language

The learner can never converge to D

Contradiction

Myhill-Nerode

**28-3 [STRE-16 Lecture 1]**

Normal testing investigates correct behavior for sensible inputs and borderline conditions. Security testing involves looking for the incorrect behavior for really silly inputs.

Why learn state machines?

Powerful analysis methods are available that can use learning models.

Very intuitive models for software systems.

TLS, printer controller, .

State machines find complexity (number of states) and deadlocks.

Automatic model checking exists (Biometric Passport, Aarts et al. 2010), (paper also contains motivation to reduce time)

**28-3 [bunq BSc. Project fsm-learner]**

Looking at the BSc. Project code (<https://github.com/bunqcom/fsm-learner>), trying to make it work. Requires other applications to be installed (brew, nodeJS, Appium, maven). Configured a virtualized environment that contained these dependencies. The BSc. Project tool has 4 options:

* 1. learn
  2. alphabet:create
  3. alphabet:compose
  4. alphabet:destroy

Before learning can be started an alphabet must be created and composed. After inspecting the code, this has been achieved by parsing XML screens.

Class com.bunq.main.Main with method runAlphabetScript on line 171 invokes a bash-script (scripts/make\_dump.sh) partly responsible of the process above. The script is not present on the public repository. After observing what the script does and attempting to create one myself, I contacted one of the developers of the BSc. Project (Tom den Braber) who shared the script via e-mail. The following is the missing script:

|  |
| --- |
| args=("$@")  FILENAME=${args[0]}  adb shell uiautomator dump  echo ${FILENAME}  adb pull /storage/emulated/legacy/window\_dump.xml alphabet/window\_dumps/$FILENAME |

*Note: The script was called with Java’s Runtime.exec() method. At this point I do not understand why the 2 commands listed in the bash-script weren’t executed like that, instead of running the script.*

A group of students also attempted to modify the code, which was documented here: <https://github.com/TUDelft-CS4110/2016-sre-crew>. Asking one of the authors who would want to make the application work again, yielded nothing useful, because the modified code wasn’t present anymore. Following their fixes proposed in the final report, resulted in a large part of the application that has been bypassed and would thus not suit the purpose of making the application work.

At this point trying to make the code work, is deemed too much of an effort with regards to the too low gain.