Title: Learning Automata of Black Box Software and Hardware with Side Channel Analysis

Author: Johnathan DiMatteo

Supervisor: Sebastian Fischmeister

Motivation

How do we verify software systems are properly designed and secure? Defending against security through obscurity. One way is to create a *model* of the software to compare to the specifications. There are numerous ways to infer models by analyzing code or performing tests (such as black box fuzzing). Black box fuzzing sends random or malformed input to a system to get the system to crash or respond in a way that does not conform to specifications. The goal is to ensure a completed system does not have unintended behaviour, which could ultimately lead to vulnerabilities. But *model learning*, also known as automata learning, is about algorithms that are able to learn a complete representation of software systems automatically. Similar to how humans learn, the algorithms provide inputs and observe outputs to determine the global states of the system. They use past behaviour of the system to select the next input to send. Over time, they can identify valid transitions and which states the system should be in to build an accurate representation (such as a state diagram or other) of the software system. With this representation, vulnerabilities should be easy to spot. Some examples of case studies where vulnerabilities were identified using model learning are: Transport Layer Security (TLS) protocols [3], Windows 8 TCP Client [4], and credit card readers [2]. As more and more companies are adopting and using IoT devices, this has become an extremely important research topic to ensure software and hardware systems conform to specifications and work as intended.

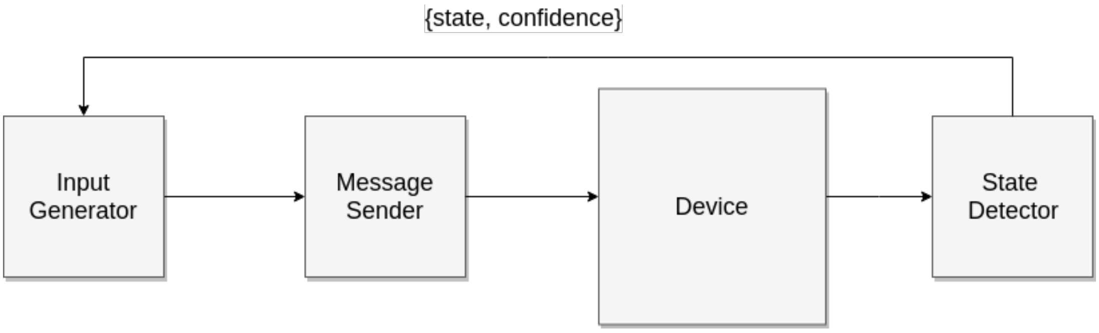
The goal of this paper is to enforce security and safety in devices through *Model Learning*. Our proposed method is to use *Side Channel Analysis* to make learning more “black box”.

Background

State diagrams are a way to model a system concerning a finite number of states, inputs and outputs. **Inputs** are external stimuli that interact with the system. **Outputs** are a system’s response to an input. A **state** determines the behavior of a system. For example, a coffee machine can have three states, “Off”, “Standby”, and “Ready”. The coffee maker will only pour coffee after the press of Button B in the “Ready” state, will be turned off in the “Off” state, and will wait for the user to load a coffee pod in the “Standby” state. A state is bound to a set of rules and functions (i.e. it is impossible to pour coffee in the “Off” state). State diagrams are common in software validation. We will be focusing on a specific type of state diagram model known as a Mealy Machine.

1 Assumptions

The first step to discover hidden functionality is to obtain the input and output alphabet of the protocol. We can do this through reverse engineering the message format of the black box, otherwise known as protocol fuzzing (cite). We will assume the input alphabet is



known (and the hidden functionality?). Reasonable because you can use protocol fuzzing to determine the input language automatically.

Any input not belonging to the known input is considering to be hidden.

Problem Statement

Given the input alphabet of a black box system, reverse engineer the FSM representing the system to find undocumented functionality.

1. **Input Generator**: Given the current predicted state (**output? - ask Sebastian**) and history of previous states (**outputs?**) and queries, determine the next query that is most likely to induce a state change in the system under observation.
2. **State Detector**: Given a response from the system under observation and history of previous responses, estimate the current state (**output? - ask Sebastian**).
3. **Message Sender**: Given a query, translate the query into a valid message accepted by the system.

Questions (and Answers) for Paper

1. framework for using powertrace for protocol fuzzing / model learning.
2. can signals be the same size? pad with zeros
3. dba averaging is effective but what is the computational complexity? is it worth it? or dtw alone.
4. can this work for model learning with embedded systems (I2C?)
5. can this work for protocol fuzzing? (web server?)
6. how does it compare to normal model learning? well, sometimes outputs aren't possible or can be duped.
7. downsample: at what sampling frequency does it still work ? show plot with time? what method did you use (averaging).
8. how does training work? is it online (batch training, record all traces and retrain every once in a while)

9. metrics for k means (parameters w, iterations)

10. how did you set up circuit (circuit diagram)

11. clustering metrics (when do you stop? which metrics did you choose?)

12. how is it better or similar to (https://arxiv.org/abs/1806.00981) Automatic Clustering of a Network Protocol with Weakly-Supervised Clustering

13. how did you train sequences of different lengths? padding? why? during classifying is it ignored? why is that okay? see http://alumni.cs.ucr.edu/ ratana/RatanamC.pdf, where they say diff lengths is ok for DTW

14. how many clusters? how did you choose k? (too many better than not enough? show plot with k)

15. what is the KB lower bound, what did you remove for DBA? different lengths sequences not allowed for KB so i had to remove it.

16. Dont forget to cite the DBA guys and the code for Dtw

References

[1] George Argyros, Ioannis Stais, Suman Jana, Angelos D Keromytis, and Aggelos Kiayias. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1690–1701. ACM, 2016.

[2] Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri De Ruiter. Automated reverse engineering using lego®. *WOOT*, 14:1–10, 2014.

[3] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., 2015. USENIX Association.

[4] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Combining model learning and model checking to analyze tcp implementations. In *International Conference on Computer Aided Verification*, pages 454–471. Springer, 2016.

[5] M. Tappler, B. K. Aichernig, and R. Bloem. Model-based testing iot communication via active automata learning. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, volume 00, pages 276–287, March 2017.

[6] Frits Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, January 2017.