Primitive Memory Application in Network Security

Capstone

by

Michael Petracca, Yevhen Voitiuk, Jeremy Pulido

1 Inspiration & Motivation

Often, the solutions to complex problems come from simple sources. Under the mentorship of Professor Michael Soltys, we explored the capabilities of bacterial memory. It has been observed that bacteria will repeat behaviors that have contributed to the survival of their predecessors. But how exactly do they "remember" something? Since bacteria are single-celled organisms, we can abstract their memory to have "primitive" form of memory that we can represent as a string of 1's and 0's.

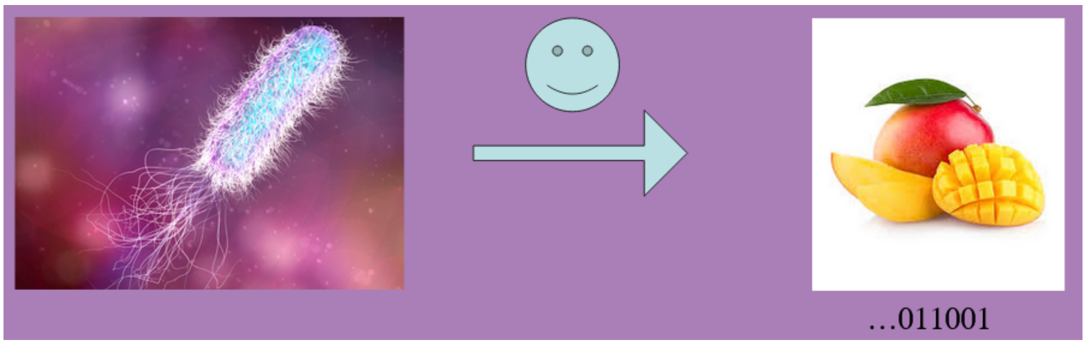


Figure 1: "Good" Event

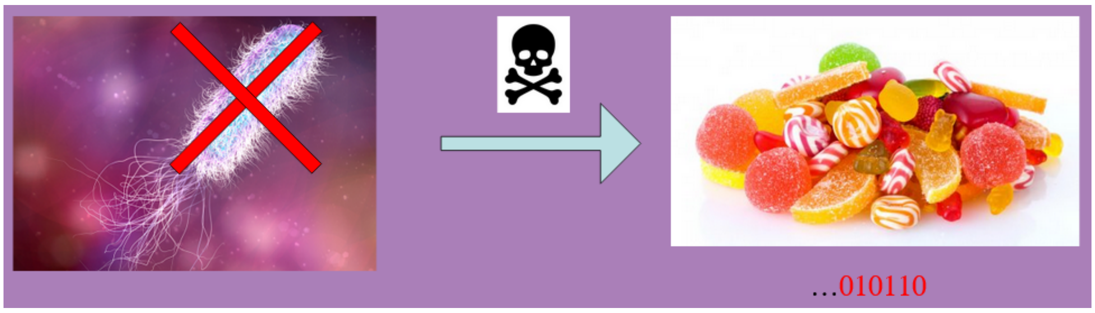


Figure 2: "Bad" Event

Above is how we conceptualize bacterial memory. When a bacteria experiences a "good" event such as eating a mango it will record this event in a form of a sub-string. As one may know, experiences can't be fully represented in just a binary "Good" or "Bad", thus an event would never be just a single binary character. Now, whenever another bacteria encounters a mango it will recall the same string and know the mango is delicious and safe to eat. However, when a bacteria consumes a piece of candy, it will, in this situation, get sick and record this event into memory with negative connotations. Thus, whenever the bacteria possessing such memory will encounter another mango, or another piece of candy, it will match the predictions from the memory and do the appropriate action.

Why are we even bothering with bacterial memory for network security? The answer is simplicity and speed. The primitive memory learning scheme we are working with can return information on the most recent matching memory swiftly.

2 Objectives & Introduction

Our main goal for this project was to do a proof of concept to showcase whether the primitive memory structure can be viable in the network security applications. In order to train and test our memory structure capabilities, we decided to create a custom simulation of the traffic light intersection.

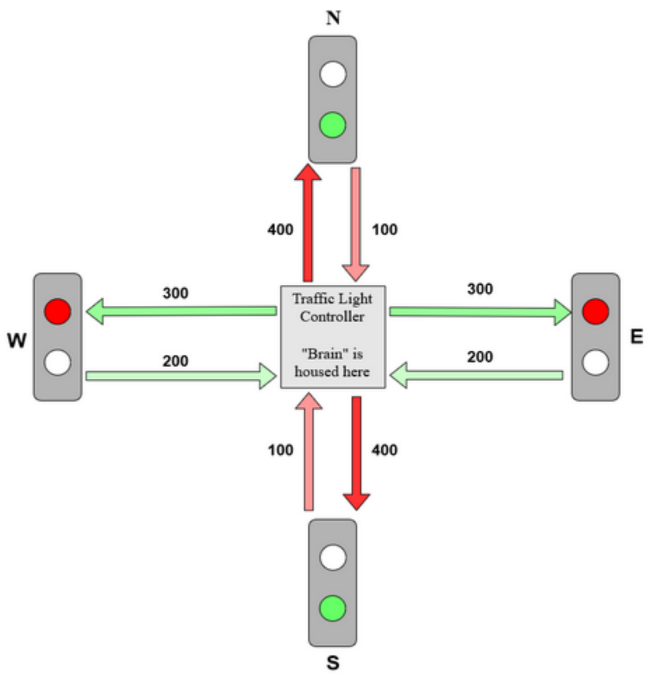


Figure 3: Traffic Light Simulation

In our simulation, we have a server/client relationship, in which we have the north (N), south (S), east (E), and west (W) stop lights communicate through a network with the traffic light controller (TLC) in order to simulate normal traffic. In normal operation, only one pair of lights should be green, while the other pair is red. So, what happens when a malicious hacker attempts to cause mayhem? Best case scenario is they make both pairs of lights red, making people late for work; worst case scenario is they make all the lights green and cause an accident.

Our memory structure will serve as an overseer for such traffic simulation. After collecting distinguishing features of both normal and abnormal network behaviors, we provide our primitive memory a model to reference, and subsequently raise an alarm when it believes the system is malfunctioning in real time. The concept is a success if our trained memory structure is able to detect abnormal behavior with above average accuracy.

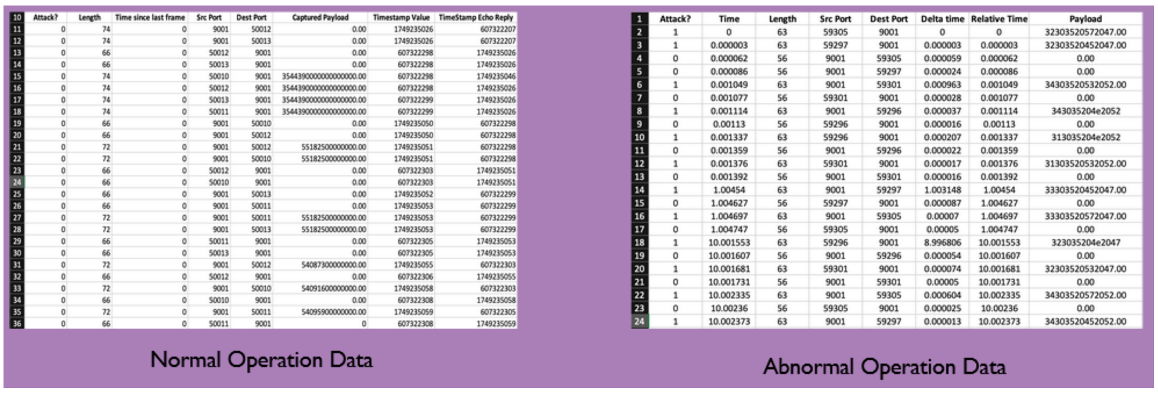
3 Methodology

We started with the implementation of the traffic light intersection simulation. Using Python 3, we implemented a client-server system in which the server (representing TLC) would facilitate the communication between four clients, the North (N), South (S), East (E), and West (W) stop lights. We then defines two modes of operation:

- Normal: Lights behave as they should
  - Only one opposing pair of lights (N and S), (E and W) can be green at a time.
  - Red lights stay up for 10 seconds, then the request is sent to change to green.
- Abnormal: One or more lights, chosen randomly by the program, request a change to green...
  - Immediately after turning red
  - Sporadically at non-standard times

The abnormal mode effectively simulates a Denial of Services (DOS) attack, both by denying the resource of being green to the adjacent lights connected to the server, as well as by using up extra server time to handle the much more frequent requests.

We then ran two different types of experiments, each with both wire shark captures (for training) and live processing of TCP traffic (for testing). The first experiment used a model produced using AWS SageMaker as a pre-processing step for the primitive memory (described below). The second experiment used data from the captures directly in the primitive memory system for the initial training.



10	Attack?	Length	Time since last frame	Src Port	Dest Port	Captured Payload	Timestamp Value	TimeStamp Echo Reply
11	0	74	0	9001	50012	0.00	1749235026	607322207
12	0	74	0	9001	50013	0.00	1749235026	607322207
13	0	66	0	50012	9001	0.00	607322298	1749235026
14	0	66	0	50013	9001	0.00	607322298	1749235026
15	0	74	0	50010	9001	35443900000000000.00	607322298	1749235046
16	0	74	0	50012	9001	35443900000000000.00	607322298	1749235026
17	0	74	0	50013	9001	35443900000000000.00	607322299	1749235026
18	0	74	0	50011	9001	35443900000000000.00	607322299	1749235026
19	0	66	0	9001	50010	0.00	1749235050	607322298
20	0	66	0	9001	50012	0.00	1749235050	607322298
21	0	72	0	9001	50012	55182500000000.00	1749235051	607322298
22	0	72	0	9001	50010	55182500000000.00	1749235051	607322298
23	0	66	0	50012	9001	0.00	607322303	1749235051
24	0	66	0	50010	9001	0.00	607322303	1749235051
25	0	66	0	9001	50013	0.00	1749235052	607322299
26	0	66	0	9001	50011	0.00	1749235053	607322299
27	0	72	0	9001	50011	55182500000000.00	1749235053	607322299
28	0	72	0	9001	50013	55182500000000.00	1749235053	607322299
29	0	66	0	50011	9001	0.00	607322305	1749235053
30	0	66	0	50013	9001	0.00	607322305	1749235053
31	0	72	0	9001	50012	54087300000000.00	1749235055	607322303
32	0	66	0	50012	9001	0.00	607322306	1749235055
33	0	72	0	9001	50010	54091600000000.00	1749235058	607322303
34	0	66	0	50010	9001	0.00	607322308	1749235058
35	0	72	0	9001	50011	54095900000000.00	1749235059	607322305
36	0	66	0	50011	9001	0	607322308	1749235059

Normal Operation Data

1	Attack?	Time	Length	Src Port	Dest Port	Delta time	Relative Time	Payload
2	1	0	63	59305	9001	0	0	32303520572047.00
3	1	0.000003	63	59297	9001	0.000003	0.000003	32303520452047.00
4	0	0.000062	56	9001	59305	0.000059	0.000062	0.00
5	0	0.000086	56	9001	59297	0.000024	0.000086	0.00
6	1	0.001049	63	9001	59301	0.000963	0.001049	34303520532052.00
7	0	0.001077	56	59301	9001	0.000028	0.001077	0.00
8	1	0.001114	63	9001	59296	0.000037	0.001114	343035204e2052
9	0	0.00113	56	59296	9001	0.000016	0.00113	0.00
10	1	0.001337	63	59296	9001	0.000207	0.001337	313035204e2052
11	0	0.001359	56	9001	59296	0.000022	0.001359	0.00
12	1	0.001376	63	59301	9001	0.000017	0.001376	31303520532052.00
13	0	0.001392	56	9001	59301	0.000016	0.001392	0.00
14	1	1.00454	63	9001	59297	1.003148	1.00454	33303520452047.00
15	0	1.004627	56	59297	9001	0.000087	1.004627	0.00
16	1	1.004697	63	9001	59305	0.00007	1.004697	33303520572047.00
17	0	1.004747	56	59305	9001	0.00005	1.004747	0.00
18	1	10.001553	63	59296	9001	8.996806	10.001553	323035204e2047
19	0	10.001607	56	9001	59296	0.000054	10.001607	0.00
20	1	10.001681	63	59301	9001	0.000074	10.001681	32303520532047.00
21	0	10.001731	56	9001	59301	0.00005	10.001731	0.00
22	1	10.002335	63	9001	59305	0.000604	10.002335	34303520572052.00
23	0	10.00236	56	59305	9001	0.000025	10.00236	0.00
24	1	10.002373	63	9001	59297	0.000013	10.002373	34303520452052.00

Abnormal Operation Data

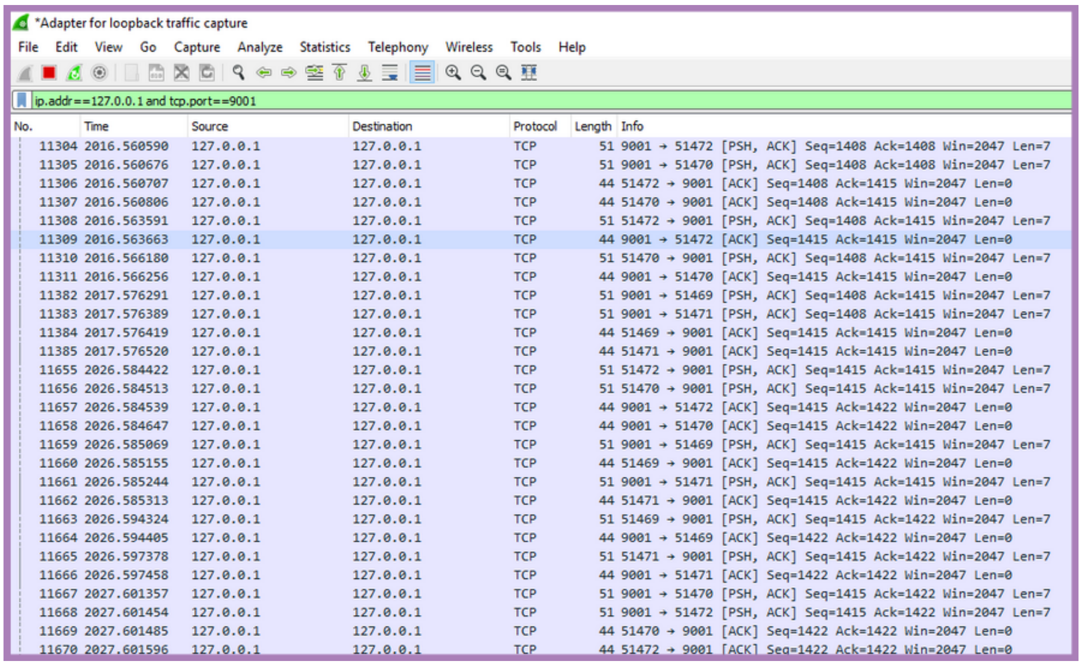
Figure 4: Training Data

We then ran both modes of operation to generate training data. We ran the simulation in two modes, on the local machine, having server and clients communicate through localhost, and through the Wireless LAN, where one PC acted as a server, and the others connected to it as clients.

The data that we focused on during the captures are as follow:

- Source and destination ports (Who was sending packets to whom)
- Length and payload of the message (What was sent)
- Packet timestamps (When was the message sent out, how long after the previous message was it send out, etc.)

This data was then filtered and captured through WireShark. On average, 10,000 packets were captured per session. There were about 1,000 frames of attack/abnormal traffic, with the remaining 9,000 frames being normal traffic. This step was the same for both experiments.


*Adapter for loopback traffic capture
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
ip.addr==127.0.0.1 and tcp.port==9001
```

No.	Time	Source	Destination	Protocol	Length	Info
11304	2016.560590	127.0.0.1	127.0.0.1	TCP	51	9001 → 51472 [PSH, ACK] Seq=1408 Ack=1408 Win=2047 Len=7
11305	2016.560676	127.0.0.1	127.0.0.1	TCP	51	9001 → 51470 [PSH, ACK] Seq=1408 Ack=1408 Win=2047 Len=7
11306	2016.560707	127.0.0.1	127.0.0.1	TCP	44	51472 → 9001 [ACK] Seq=1408 Ack=1415 Win=2047 Len=0
11307	2016.560806	127.0.0.1	127.0.0.1	TCP	44	51470 → 9001 [ACK] Seq=1408 Ack=1415 Win=2047 Len=0
11308	2016.563591	127.0.0.1	127.0.0.1	TCP	51	51472 → 9001 [PSH, ACK] Seq=1408 Ack=1415 Win=2047 Len=7
11309	2016.563663	127.0.0.1	127.0.0.1	TCP	44	9001 → 51472 [ACK] Seq=1415 Ack=1415 Win=2047 Len=0
11310	2016.566180	127.0.0.1	127.0.0.1	TCP	51	51470 → 9001 [PSH, ACK] Seq=1408 Ack=1415 Win=2047 Len=7
11311	2016.566256	127.0.0.1	127.0.0.1	TCP	44	9001 → 51470 [ACK] Seq=1415 Ack=1415 Win=2047 Len=0
11382	2017.576291	127.0.0.1	127.0.0.1	TCP	51	9001 → 51469 [PSH, ACK] Seq=1408 Ack=1415 Win=2047 Len=7
11383	2017.576389	127.0.0.1	127.0.0.1	TCP	51	9001 → 51471 [PSH, ACK] Seq=1408 Ack=1415 Win=2047 Len=7
11384	2017.576419	127.0.0.1	127.0.0.1	TCP	44	51469 → 9001 [ACK] Seq=1415 Ack=1415 Win=2047 Len=0
11385	2017.576520	127.0.0.1	127.0.0.1	TCP	44	51471 → 9001 [ACK] Seq=1415 Ack=1415 Win=2047 Len=0
11655	2026.584422	127.0.0.1	127.0.0.1	TCP	51	51472 → 9001 [PSH, ACK] Seq=1415 Ack=1415 Win=2047 Len=7
11656	2026.584513	127.0.0.1	127.0.0.1	TCP	51	51470 → 9001 [PSH, ACK] Seq=1415 Ack=1415 Win=2047 Len=7
11657	2026.584539	127.0.0.1	127.0.0.1	TCP	44	9001 → 51472 [ACK] Seq=1415 Ack=1422 Win=2047 Len=0
11658	2026.584647	127.0.0.1	127.0.0.1	TCP	44	9001 → 51470 [ACK] Seq=1415 Ack=1422 Win=2047 Len=0
11659	2026.585069	127.0.0.1	127.0.0.1	TCP	51	9001 → 51469 [PSH, ACK] Seq=1415 Ack=1415 Win=2047 Len=7
11660	2026.585155	127.0.0.1	127.0.0.1	TCP	44	51469 → 9001 [ACK] Seq=1415 Ack=1422 Win=2047 Len=0
11661	2026.585244	127.0.0.1	127.0.0.1	TCP	51	9001 → 51471 [PSH, ACK] Seq=1415 Ack=1415 Win=2047 Len=7
11662	2026.585313	127.0.0.1	127.0.0.1	TCP	44	51471 → 9001 [ACK] Seq=1415 Ack=1422 Win=2047 Len=0
11663	2026.594324	127.0.0.1	127.0.0.1	TCP	51	51469 → 9001 [PSH, ACK] Seq=1415 Ack=1422 Win=2047 Len=7
11664	2026.594405	127.0.0.1	127.0.0.1	TCP	44	9001 → 51469 [ACK] Seq=1422 Ack=1422 Win=2047 Len=0
11665	2026.597378	127.0.0.1	127.0.0.1	TCP	51	51471 → 9001 [PSH, ACK] Seq=1415 Ack=1422 Win=2047 Len=7
11666	2026.597458	127.0.0.1	127.0.0.1	TCP	44	9001 → 51471 [ACK] Seq=1422 Ack=1422 Win=2047 Len=0
11667	2027.601357	127.0.0.1	127.0.0.1	TCP	51	9001 → 51470 [PSH, ACK] Seq=1415 Ack=1422 Win=2047 Len=7
11668	2027.601454	127.0.0.1	127.0.0.1	TCP	51	9001 → 51472 [PSH, ACK] Seq=1415 Ack=1422 Win=2047 Len=7
11669	2027.601485	127.0.0.1	127.0.0.1	TCP	44	51470 → 9001 [ACK] Seq=1422 Ack=1422 Win=2047 Len=0
11670	2027.601596	127.0.0.1	127.0.0.1	TCP	44	51472 → 9001 [ACK] Seq=1422 Ack=1422 Win=2047 Len=0

Figure 5: WireShark Capture

The first experiment proceeded as follows:

1. Captured 10,000 frames of traffic in WireShark.
2. This was then parsed using Python and then sent to AWS SageMaker as training data.
3. Using the Linear Learner scheme (with binary classification mode), we acquired a trained model object.
4. The classification model was then run locally to train the primitive memory model.
5. The total system was then run using live input from an active simulation, producing a prediction.

The second experiment proceeded as follows:

1. Captured 10,000 frames of traffic in WireShark.
2. Packet information was parsed using Python and sent into the primitive memory model as training data
3. The primitive memory model was then taught to classify the packet strings.
4. The resulting trained memory model was run on live input from an active simulation, producing a prediction.

4 Conclusions

After gathering and comparing the results from both experiments, it's clear to say that the memory model was able to perform much better when it was aided with predetermined data from the SageMaker model. Using SageMaker provided a basis for our primitive memory to work with, already having an encoding for a "good" or "bad" event. When our system had no such model to work with it essentially guessed what was "good" or "bad", forcing the results to be more random. Alternatively, the system can notify an administrator of any decisions it is unsure of, and the administrator's responses can be used to train the model.

Due to the potential labor intensiveness of manual training/decision review, it is still helpful to employ a separate machine learning algorithm to classify the input in order to seed the system. Otherwise, it is necessary to bootstrap the system by manually entering classification, and providing a memory string file and associated decision file for the BactMem constructor to make use of.

Thus, a primitive memory model like ours definitely shows potential in applications of network security. However, more diverse testing has to be done with different machine learning models and complex systems than a traffic light intersection.

It is important to note that, with the current implementation, highly specific data could potentially confound the system, by preventing similar memories from matching. This could be avoided in data selection, such as providing some kind of metadata instead of the actual data. Also, this may limit the application of this model only to simpler data sets, unless it is used in tandem with another system, as mentioned above.

5 Future Research

It is clear that much more extensive research and experimentation are needed in order to cement the value of the primitive memory model for network security purposes. As the topic can be achieved using vastly different methodologies, we compiled the list of improvements and additional functionalities for our approach to this issue that could potentially improve the results and the reputation of the primitive memory model in Computer Science.

The proposed improvements are:

1. Hosting the server on different web services and connecting clients to it from different machines.
    - a Consider using VPNs, the capture data will be more diverse
    - b Will help determine more distinguishing features for our machine learning models
        - i IP, sequence numbers, IPv4 protocol information could also then hold significant data.
2. The simulation then could be expanded to handle more different types of attacks that don't necessarily happen exclusively through the network in order to test the model's capability to infer the attack from observing the absence of proper network traffic, rather than detecting explicitly malicious packets.
3. Considering the memory structure, more research into data selection and pre-processing would be beneficial. By combating the previously mentioned shortcomings, we could improve the performance of the system significantly.
4. Finally, the response kit could be constructed such that it works together with our observing memory structure in order to properly respond to the incident opposed to simply raising alert flags when the malicious event is matched.

6 Resources

- AWS
- Amazon SageMaker
- Python
- TCPDump
- WireShark
- Herman, Grzegorz, and Michael Soltys. “On the Ehrenfeucht–Mycielski Sequence.” Journal of Discrete Algorithms, vol. 7, no. 4, 2009, pp. 500–508., doi:10.1016/j.jda.2009.01.002.