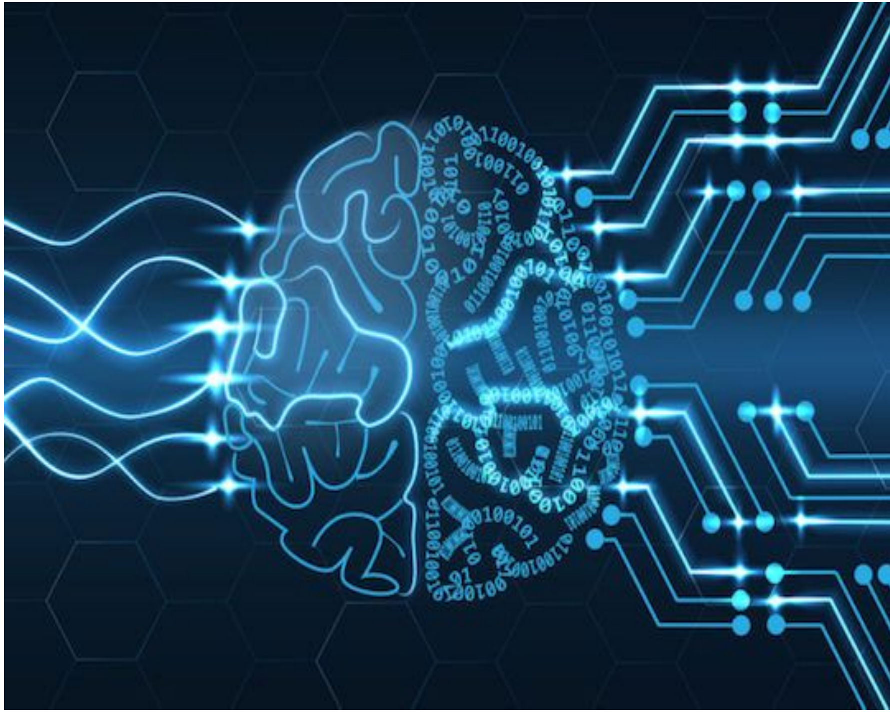
Line Coverage Improvement of Objdump using AFL Fuzzing and Neural Networks

By Yuval Schaal and Ido Michael

Motivation

- As computer engineering students, we are interested in machine learning.
- We wanted to use machine learning to dive into software reliability.
- Test if there was a way to make software more reliable using ML.



Project layout

1. Learning the basics and documentation of AFL fuzzer, instrument into a simple hello world.
2. Generate another hello world with Gcov & Lcov instrumentations and fuzz hello world app to generate seeds.
3. Work on a real world program such as binutils and generate training/testing data for the network (fuzzing seeds), Fuzzing compiled objects with a simple small input to mutate it and get better coverage.
4. Train the Neural Network with this data.
5. Testing the code coverage with Lcov, to see the unique crashes and line coverage.

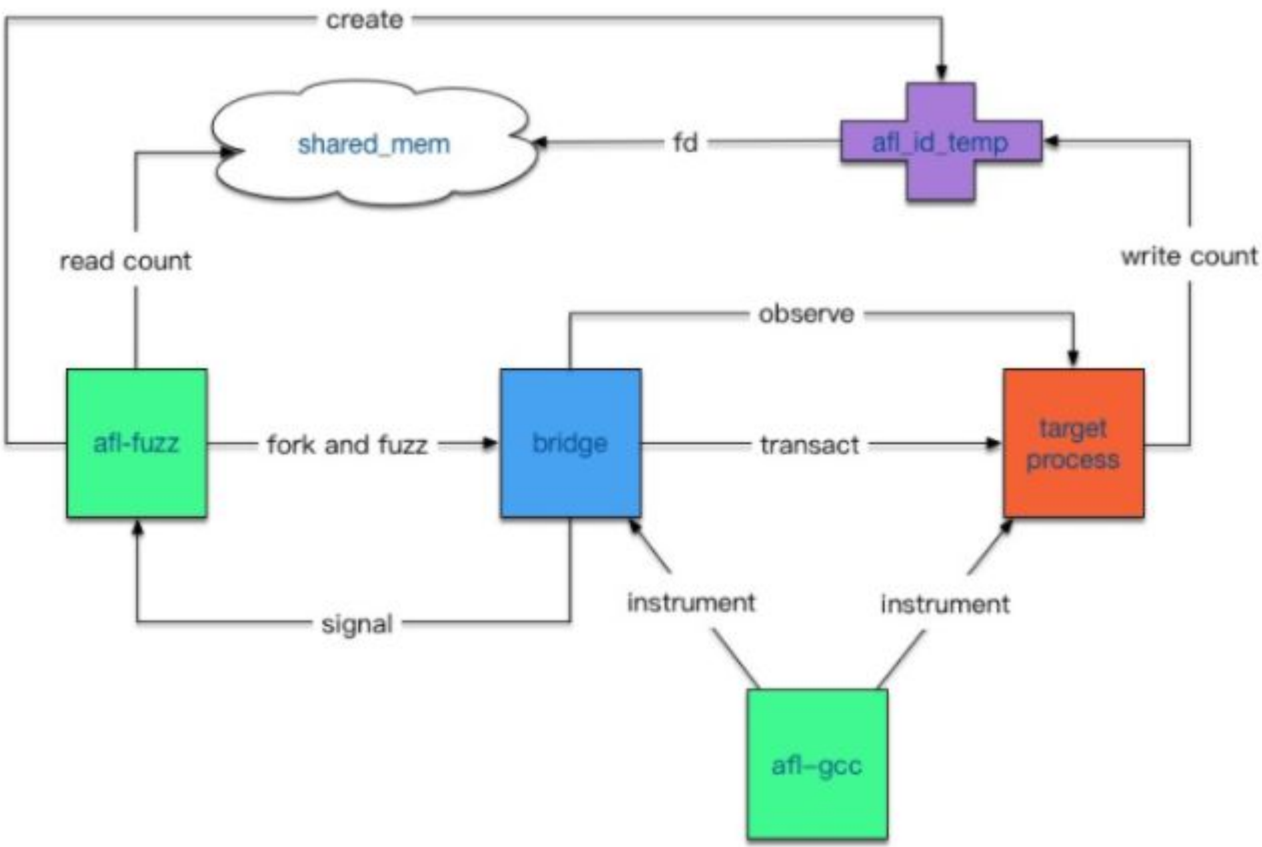
Previous Work

- Our mentor for this project is Dongdong She, a current Columbia CS PHD student of Prof. Junfeng.
- His work, very recently done, was to run the coverage project on readelf, which displays information about one or more ELF format object files, and see how to get better line coverage for software reliability.

Introduction to AFL Fuzzing

1. Compiling a binary with AFL instrumentations
2. Fuzzing those compiled objects with a simple small input to mutate it, all to get a better code coverage.
3. Testing the code coverage with Lcov, to see the unique crashes and line coverage.
4. AFL also contains a tool called afl-showmap, we will review it during the bitmap generation.

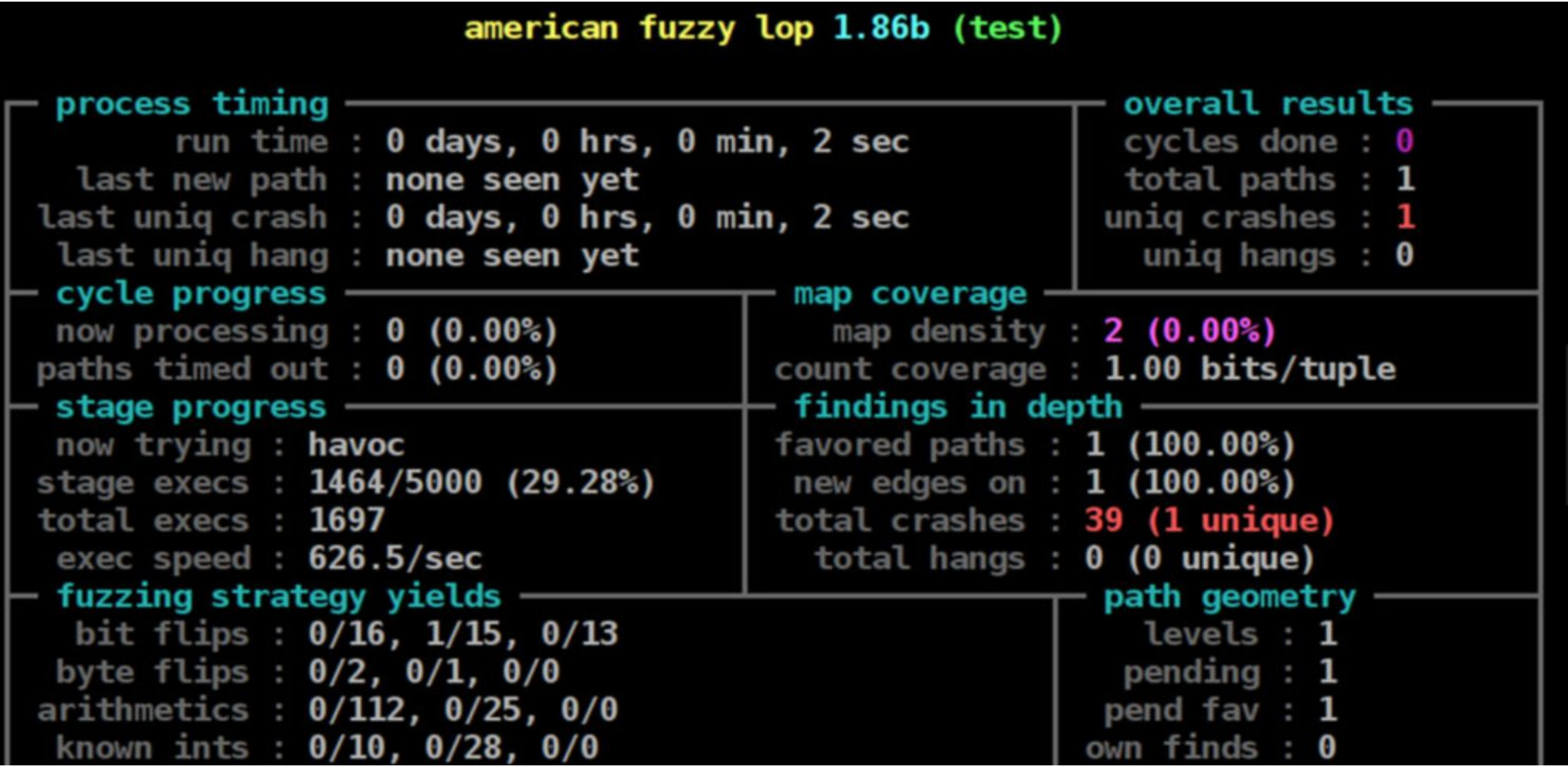
AFL Fuzzing diagram



Fuzzing process

- The AFL has a file called afl-gcc: this is the adapter for gcc compiler that creates the instrumentation into the obj file.
- After instrumenting, we need to fuzz the compiled file with some input and flag (depends on the program). This fuzzing is mutating the input file to get to new functions and get more coverage.
- We ran 8 AFL threads which synchronise once in a while, we got a total of about 6K seeds as training data and 8K for testing, total of 14K seeds.

How it actually looks like?



Background Objdump

- After Dongdong had successful results with Readelf, we decided to test a smaller, similar file: Objdump, also a part of the GNU Binary Utilities.
- Objdump is a program for displaying various information about object files on Unix-like systems. For instance, it can be used as a disassembler to view an executable in assembly form.
- During the process we have also used this tool to check if the compiled object files contained gcov/afl instrumentations.
- There are multiple flags you can use with Objdump, -D causes the highest function coverage which is why we chose to use it.

How Objdump Looks Like?

```
[root@linux-server root]#
[root@linux-server root]# objdump -D a.out | grep -A20 main.:
08048460 <main>:
 8048460:       55                      push   %ebp
 8048461:       89 e5                   mov    %esp,%ebp
 8048463:       83 ec 08                sub    $0x8,%esp
 8048466:       90                      nop
 8048467:       c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
 804846e:       89 f6                   mov    %esi,%esi
 8048470:       83 7d fc 09             cmpl   $0x9,0xfffffffc(%ebp)
 8048474:       7e 02                   jle    8048478 <main+0x18>
 8048476:       eb 18                   jmp    8048490 <main+0x30>
 8048478:       83 ec 0c                sub    $0xc,%esp
 804847b:       68 08 85 04 08          push   $0x8048508
 8048480:       e8 93 fe ff ff          call   8048318 <_init+0x38>
 8048485:       83 c4 10                add    $0x10,%esp
 8048488:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
 804848b:       ff 00                   incl   (%eax)
 804848d:       eb e1                   jmp    8048470 <main+0x10>
 804848f:       90                      nop
 8048490:       b8 00 00 00 00          mov    $0x0,%eax
 8048495:       c9                      leave
 8048496:       c3                      ret
[root@linux-server root]# _
```

Background - Gcov

- A source code coverage analysis and statement-by-statement profiling tool
- Generates exact counts of the number of times each statement in a program is executed and annotates source code to add instrumentation

```c
#include <stdio.h>

int
main (void)
{
  int i;

  for (i = 1; i < 10; i++)
    {
      if (i % 3 == 0)
        printf ("%d is divisible by 3\n", i);
      if (i % 11 == 0)
        printf ("%d is divisible by 11\n", i);
    }

  return 0;
}
```

C program before running gcov

```
            #include <stdio.h>

            int
            main (void)
            {
     1        int i;

    10        for (i = 1; i < 10; i++)
                {
     9            if (i % 3 == 0)
     3              printf ("%d is divisible by 3\n", i);
     9            if (i % 11 == 0)
######              printf ("%d is divisible by 11\n", i);
     9          }

     1        return 0;
     1      }
```

Gcov file

Background - Lcov

1. LCOV is a graphical front-end for GCC's coverage testing tool gcov.
2. After generating all of the necessary files we needed to run Lcov to create a HTML summary of the code coverage and the actual crashes, unique functions etc.

LCOV - code coverage report

			Hit	Total
Current view:	top level			
Test:	libbash test coverage	**Lines:**	20640	34749
Date:	2011-05-26	**Functions:**	1184	1287
		Branches:	15689	37086

Directory	Line Coverage			Functions	
src/core		95.7 %	314 / 328	98.2 %	55 / 56
test		97.0 %	98 / 101	100.0 %	72 / 72
src/builtins/tests		98.6 %	144 / 146	100.0 %	203 / 203
src/builtins		98.6 %	214 / 217	100.0 %	45 / 45
src/core/tests		98.9 %	351 / 355	99.3 %	133 / 134
./src/builtins		100.0 %	9 / 9	93.3 %	14 / 15
src		100.0 %	35 / 35	91.7 %	11 / 12
./src/core		100.0 %	190 / 190	98.0 %	99 / 101

Generated by: LCOV version 1.9

Background - Keras and Tensorflow

- Tensorflow is an open source software library, for this project we used it for the machine learning applications utilizing neural networks.
- Keras runs on top of Tensorflow and is designed to enable fast experimentation with deep neural networks.

Environment

- Google Cloud Platform
- Compute Engine VM Instance
- GPU NVIDIA Tesla K80
- 8 Core CPU
- 500 GB Standard persistent disk
- Ubuntu 14.04

Generating the bitmap data

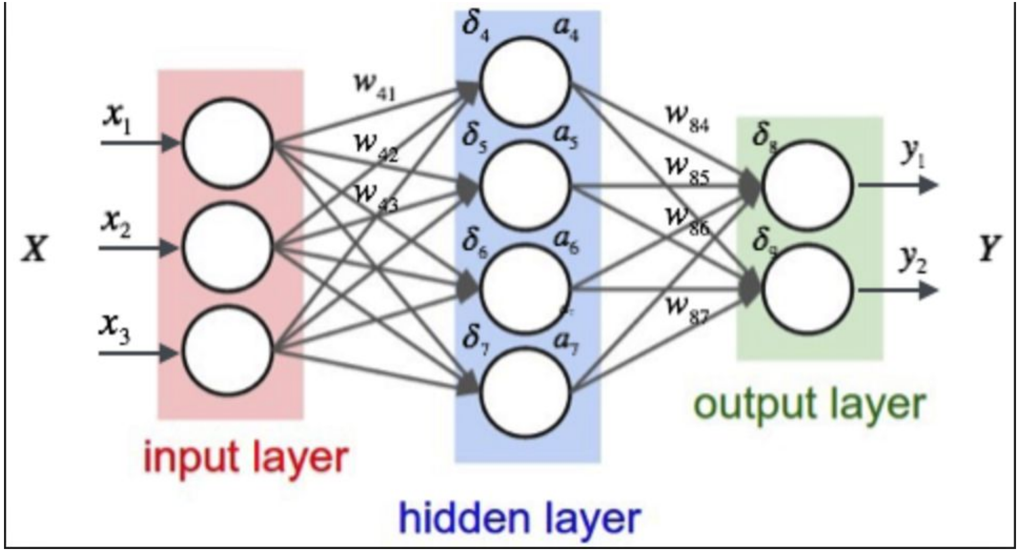
- As we mentioned the AFL has a tool called showmap.
- This tool runs the targeted binary and displays the contents of the trace bitmap in a human-readable form It’s very useful in scripts to eliminate redundant inputs and perform other checks.
- We have fed to this tool the compiled binary with the mutated seeds that we generated, to create bitmaps (our NN data).

Cleaning the data

- We removed any seeds which was larger than 10K.
- Then we needed to put all the unique bitmaps of both the training and testing (fuzzer01, fuzzer02) into a single file:
    - `awk '{if (!a[$0]++) print}' ./fuzzer01/bitmaps/* > tempfile`
    - `awk '{if (!a[$0]++) print}' ./fuzzer02/bitmaps/* >> tempfile`
- After that we used a python script: convert_bitmap.py to create both the convert_bitmaps folder, as well as the convert_testing_bitmaps.

Training the model

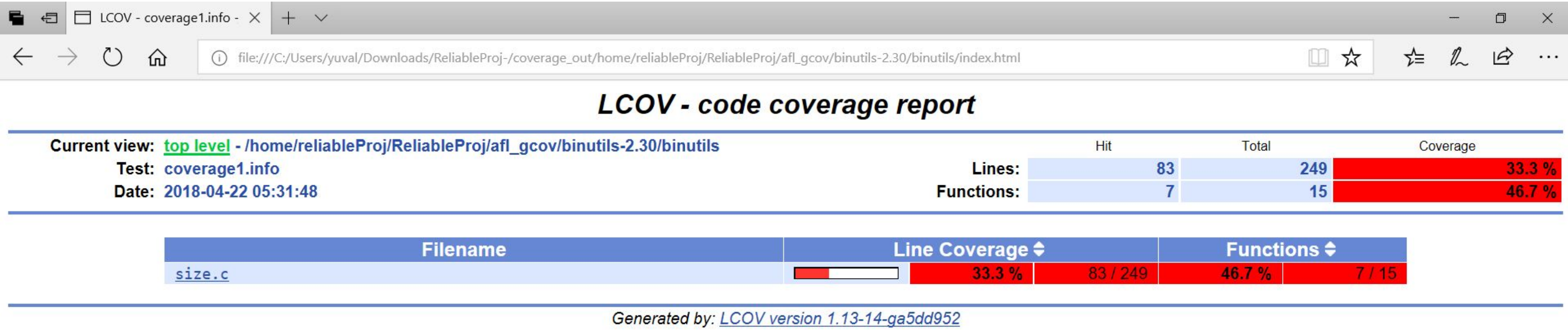
- To train the model we used four layers in the neural network with 70 epochs, and 144 steps per each epoch.
- The results of this model had 97.25% accuracy.



Running gcov on the NN output

- So far, we got up to this step and we have the outputs.
- After running the NN and Mutate_gcov we get millions of mutated seeds.
- This step includes running those seeds on the gcov files one by one to create coverage report.
- With this report we can see if we got better results using the NN.
- Lcov will generate a html file that will be a report of the line coverage for objdump

Lcov results



LCOV - coverage1.info -

file:///C:/Users/yuval/Downloads/ReliableProj-/coverage_out/home/reliableProj/ReliableProj/afl_gcov/binutils-2.30/binutils/index.html

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - /home/reliableProj/ReliableProj/afl_gcov/binutils-2.30/binutils				
Test:	coverage1.info	**Lines:**	83	249	33.3 %
Date:	2018-04-22 05:31:48	**Functions:**	7	15	46.7 %

Filename	Line Coverage			Functions	
size.c		33.3 %	83 / 249	46.7 %	7 / 15

Generated by: LCOV version 1.13-14-ga5dd952

Conclusions

1. Fuzzing is a really powerful tool on it’s own.
2. Fuzzing and Neural Networks can shorten and even eliminate the current debugging process of software development.
3. All of the pipeline of fuzzing the seeds to generate data, creating bitmaps out of those seeds, training the NN, feeding millions of seeds to Gcov, has a really high complexity, it took us about 3 days to run end-to-end with a powerful GPU.

Future work

1. There are many improvement and changes that can be made, running on different programs, changing the data amounts or parameters, trying different network architectures.
2. A continuous, automated process can be achieved with LSTMs (Long-short term memory), training a model and once in a while feeding it with new seeds that the fuzzer has generated.
3. Because of the heavy complexity of the pipeline we didn’t really meet a limit of the fuzzer/the network. An interesting experiment would be to test the limits of the system and see when the effectiveness drops.

References

Rawat, Sanjay, and Vivek Jain. VUzzer: Application-Aware Evolutionary Fuzzing. (https://www.cs.vu.nl/~herbertb/download/papers/vuzzer_ndss17.pdf)

Prashant Anantharaman, Michael C. Millian, Sergey Bratus, Meredith L. Patterson, "Input Handling Done Right: Building Hardened Parsers Using Language-Theoretic Security", Cybersecurity Development (SecDev) 2017 IEEE, pp. 4-5, 2017 (https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7839812)

Rick, et al. “Complementing Model Learning with Mutation-Based Fuzzing.” [1611.02429] Complementing Model Learning with Mutation-Based Fuzzing, 8 Nov. 2016, arxiv.org/abs/1611.02429. (https://arxiv.org/abs/1611.02429)

Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, Fu Song, "SPAIN: security patch analysis for binaries towards understanding the pain and pills", Software Engineering Proceedings of the 39th International Conference (https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7958599)

Thanks!

Any questions?

You can find us at

im2492@columbia.edu & ys3055@columbia.edu