Getting started with vulnerability discovery using Machine Learning

by

G. Grieco

Objectives

1. Theory
    1.1 Understand how Machine Learning is used ...
    1.2 and how you can apply i it for vulnerability detection.
2. Practice:
    2.1 Prediction and test case visualization with VDiscover

Motivation and previous work

Vulnerabilities in software 101

```c
int main (int argc, char *argv[]) {
  char user[128];
  char cmd[128];
  char buffer[1024];

  strcpy(cmd, "./show_users");
  strcpy(user, argv[1]);

  FILE* stream = popen(cmd, "r");
  fread (buffer, 1, 1023, stream);
  buffer[1023] = NULL;

  if (strstr(buffer, user)) {
    printf("user \"%s\" found\n", user);
  }

  pclose(stream);
  return 0;
}
```

Normally:

prog “root” → user "root" found

Unfortunately:

prog “aaa...”→ sh: 1: aaa...: not found

Vulnerabilities in software 101

```
int main (int argc, char *argv[]) {
  char user[128];
  char cmd[128];
  char buffer[1024];

  strcpy(cmd, "./show_users");
  strcpy(user, argv[1]);

  FILE* stream = popen(cmd, "r");
  fread (buffer, 1, 1023, stream);
  buffer[1023] = NULL;

  if (strstr(buffer, user)) {
    printf("user \"%s\" found\n", user);
  }

  pclose(stream);
  return 0;
}
```

Normally:

prog “root” → user "root" found

Unfortunately:

prog “aaa...”→ sh: 1: aaa...: not found

Vulnerabilities in software 101

```
int main (int argc, char *argv[]) {
  char user[128];
  char cmd[128];
  char buffer[1024];

  strcpy(cmd, "./show_users");
  strcpy(user, argv[1]);

  FILE* stream = popen(cmd, "r");
  fread (buffer, 1, 1023, stream);
  buffer[1023] = NULL;

  if (strstr(buffer, user)) {
    printf("user \"%s\" found\n", user);
  }

  pclose(stream);
  return 0;
}
```

Normally:

prog “root” $\rightarrow$ user "root" found

Unfortunately:

prog “aaa...”$\rightarrow$ sh: 1: aaa...: not found

Vulnerabilities in software 102

1. **Static Program Analysis**: to scan code to detect potential vulnerabilities
   - **Imprecise**, since we don't know how the code will execute.
2. **Dynamic Program Analysis**: to check a program behavior to detect vulnerabilities
   - **Incomplete**, since we can't examine all the possible executions.
3. **Testing**: to fuzz a normal input ("user" $\rightarrow$ {"uer","uuuser", ...})
   - **Not so effective**: it could require many mutations to uncover interesting bugs.

The problem:

- For every program and vulnerability to discovery, these approaches are quite **costly** in terms of computation power.
- We have a **large number programs** and **bugs reports** to analyze (e.g, in Debian we have more than 30k programs and 80k bug reports)

Vulnerabilities in software 102

1. **Static Program Analysis**: to scan code to detect potential vulnerabilities
   - **Imprecise**, since we don't know how the code will execute.
2. **Dynamic Program Analysis**: to check a program behavior to detect vulnerabilities
   - **Incomplete**, since we can't examine all the possible executions.
3. **Testing**: to fuzz a normal input ("user" $\rightarrow$ {"uer","uuuser", ...})
   - **Not so effective**: it could require many mutations to uncover interesting bugs.

The problem:

- For every program and vulnerability to discovery, these approaches are quite **costly** in terms of computation power.
- We have a **large number programs** and **bugs reports** to analyze (e.g, in Debian we have more than 30k programs and 80k bug reports)

Introducing traces

- Sequences of program events:
  - Calls to standard functions and its arguments (strcpy, memcpy, malloc, free, ..)
  - Final state (exit, crash, abort or timeout)

- prog "root"
  1. strcpy
  2. strcpy
  3. popen
  4. fread
  5. strstr
  6. printf
  7. pclose
  8. **exit**

- prog "roAAAAot"
  1. strcpy
  2. strcpy
  3. popen
  4. fread
  5. strstr
  6. pclose
  7. **exit**

- prog
  1. strcpy
  2. strcpy..
  3. **crash**

What we want?

- A technique to **automatize** or **assist vulnerability discovery** from traces extracted directly from **binaries.**

Key ideas:

- We have a **large** number of programs/test cases to analyze.
- We want to learn patterns in a **small** sample of analyzed programs/test cases.
- We would like to predict which **new** programs/test cases are more likely to uncover vulnerabilities.

What we want?

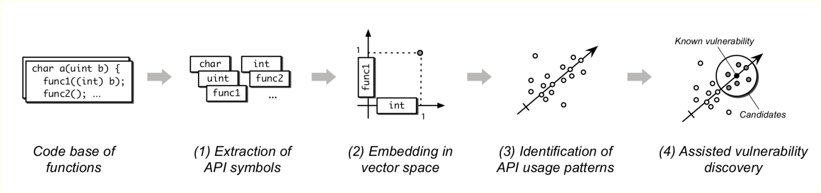
- A technique to **automatize** or **assist vulnerability discovery** from traces extracted directly from **binaries.**

Key ideas:

- We have a **large** number of programs/test cases to analyze.
- We want to learn patterns in a **small** sample of analyzed programs/test cases.
- We would like to predict which **new** programs/test cases are more likely to uncover vulnerabilities.

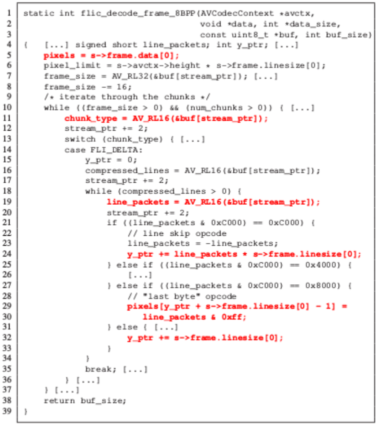
Previous Work

Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning [5]



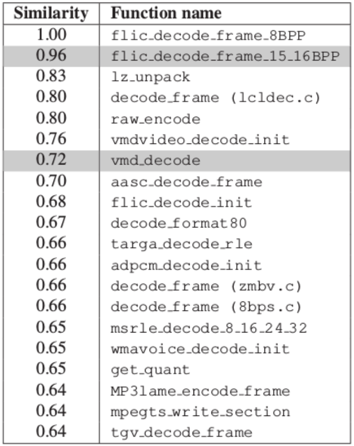
char a(uint b) {
func1((int) b);
func2(); ...
char
int
uint
func2
func1
...
1
func1
int
1
Known vulnerability
Candidates
Code base of functions
(1) Extraction of API symbols
(2) Embedding in vector space
(3) Identification of API usage patterns
(4) Assisted vulnerability discovery

Previous Results



```c
static int flic_decode_frame_8BPP(AVCodecContext *avctx,
                                  void *data, int *data_size,
                                  const uint8_t *buf, int buf_size)
{   [...] signed short line_packets; int y_ptr; [...]
    pixels = s->frame.data[0];
    pixel_limit = s->avctx->height * s->frame.linesize[0];
    frame_size = AV_RL32(&buf[stream_ptr]); [...]
    frame_size -= 16;
    /* iterate through the chunks */
    while ((frame_size > 0) && (num_chunks > 0)) { [...]
        chunk_type = AV_RL16(&buf[stream_ptr]);
        stream_ptr += 2;
        switch (chunk_type) { [...]
        case FLI_DELTA:
            y_ptr = 0;
            compressed_lines = AV_RL16(&buf[stream_ptr]);
            stream_ptr += 2;
            while (compressed_lines > 0) {
                line_packets = AV_RL16(&buf[stream_ptr]);
                stream_ptr += 2;
                if ((line_packets & 0xC000) == 0xC000) {
                    // line skip opcode
                    line_packets = -line_packets;
                    y_ptr += line_packets * s->frame.linesize[0];
                } else if ((line_packets & 0xC000) == 0x4000) {
                    [...]
                } else if ((line_packets & 0xC000) == 0x8000) {
                    // "last byte" opcode
                    pixels[y_ptr + s->frame.linesize[0] - 1] =
                       line_packets & 0xff;
                } else { [...]
                    y_ptr += s->frame.linesize[0];
                }
            }
            break; [...]
        } [...]
    } [...]
    return buf_size;
}
```

Figure 4: Original vulnerability (CVE-2010-3429).



Similarity	Function name
1.00	flic_decode_frame_8BPP
0.96	flic_decode_frame_15_16BPP
0.83	lz_unpack
0.80	decode_frame (lcldec.c)
0.80	raw_encode
0.76	vmdvideo_decode_init
0.72	vmd_decode
0.70	aasc_decode_frame
0.68	flic_decode_init
0.67	decode_format80
0.66	targa_decode_rle
0.66	adpcm_decode_init
0.66	decode_frame (zmbv.c)
0.66	decode_frame (8bps.c)
0.65	msrle_decode_8_16_24_32
0.65	wmavoice_decode_init
0.65	get_quant
0.64	MP3lame_encode_frame
0.64	mpegts_write_section
0.64	tgv_decode_frame

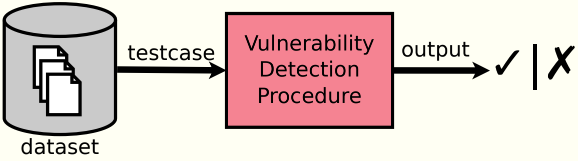
The plan

In general:

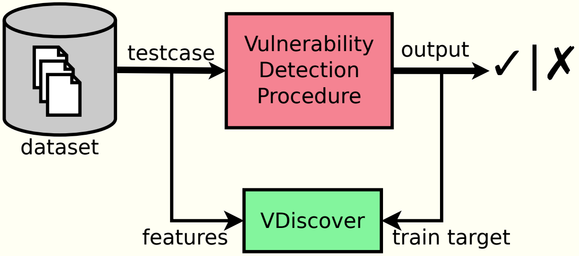
1. Data collection
2. Embedding in a *n*-dimensional space
3. Model training/inference
4. Testing the trained model with new data

Toward large-scale vulnerability discovery using Machine Learning

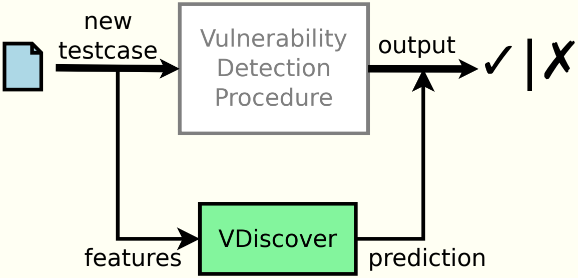
Overview: Training



Overview: Training



Overview: Training



Key Principles of VDiscover

1. **No source-code required**: Our features are extracted using static and dynamic analysis for binaries programs, allowing our technique to be used in proprietary operating systems.
2. **Automation**: No human intervention is need to select features to predict, we focused only on feature sets that can be extracted and selected automatically, given a large enough dataset.
3. **Scalability**: Since we want to focus on scalable techniques, we only use lightweight static and dynamic analysis. Costly operations like instruction per instruction reasoning are avoided by design.

Data collection

Initial setup

1. Download and extract the Mayhem bugs [3] from the Debian Bug Tracker [2].
   - A total of 1039 bugs in 496 packages.
   - Every bug is packed with a crash report and the required inputs to reproduce it.
2. Prepare a virtual machine (Vagrant box [4]) and install the required packages.

Let's take look to a small test case..

Reproducible test cases

- xa is a small cross-assembler for the 65xx series of 8-bit processors (i.e. Commodore 64). A test case is structured like this:

path.txt	/usr/bin/xa	
crash		
	argv_1.symb	\bo@e\0
	argv_2.symb	@o
	argv_3.symb	-o

This bug is *not* fixed

```
$ gdb --args env -i /usr/bin/xa '\bo@e\0' '@o' '-o'
...
Program received signal SIGSEGV, Segmentation fault.
(gdb) x/i $eip => 0x8049788:  movzbl (%ecx),%eax
(gdb) info registers
eax 0x0 0
ecx 0x0 0
...
```

Question:

Should we spend our resources trying to fuzz this test case?

This vulnerability is not fixed!

```
$ gdb --args env -i /usr/bin/xa '\bo@e\0' '@o' 'AAAA...AAAA-o'
```

```
Copyright (C) 1989-2009 Andre Fachat, Jolse Maginnis, David
Weinehall
o@e:line 1:  1000:Syntax error
and Cameron Kaiser.
o@e:line 2:  1000:Syntax error
Couldn't open source file '@o'!
o@e:line 3:  1000:Syntax error
Couldn't open source file 'o@'!
*** buffer overflow detected ***:  /usr/bin/xa terminated
...
```

This vulnerability is not fixed!

```
$ gdb --args env -i /usr/bin/xa '\bo@e\0' '@o' 'AAAA...AAAA-o'

Copyright (C) 1989-2009 Andre Fachat, Jolse Maginnis, David
Weinehall
o@e:line 1:  1000:Syntax error
and Cameron Kaiser.
o@e:line 2:  1000:Syntax error
Couldn't open source file '@o'!
o@e:line 3:  1000:Syntax error
Couldn't open source file 'o@'!
*** buffer overflow detected ***:  /usr/bin/xa terminated
...
```

A predictive approach

- We want to predict if our fuzzer will find at least **one** interesting memory vulnerability (e.g, stack corruption). These cases will be **flagged** for further analysis.

- To train, we need to collect useful data from every test case available. This data are our **features.**

Question:

Which features we can use to train? (yes, traces, but how?)

A small fragment of the xa assembly

```
 1  call getenv
 2  test %eax,%eax
 3  je @11
 4  lea −0x100c(%ebp),%ebx
 5  mov %eax,0x4(%esp)
 6  mov %ebx,(%esp)
 7  call strcpy
 8  movl $0x123,0x4(%esp)
 9  mov  %ebx,(%esp)
10  call strtok
11  ...
12  ret
```

Standard C library functions are one of the **basic blocks** in low level programing!

A small fragment of the xa assembly

```
 1  call getenv
 2  test %eax,%eax
 3  je @11
 4  lea -0x100c(%ebp),%ebx
 5  mov %eax,0x4(%esp)
 6  mov %ebx,(%esp)
 7  call strcpy
 8  movl $0x123,0x4(%esp)
 9  mov  %ebx,(%esp)
10  call strtok
11  ...
12  ret
```

Standard C library functions are one of the **basic blocks** in low level programing!

Tracing xa

ltrace	**VDiscover**
getenv('XAINPUT')	getenv(GPtr32)
strcpy(0xbfffc0fc, input)	strcpy(SPtr32,HPtr32)
strtok('input', ',')	strtok(HPtr32,GPtr32)

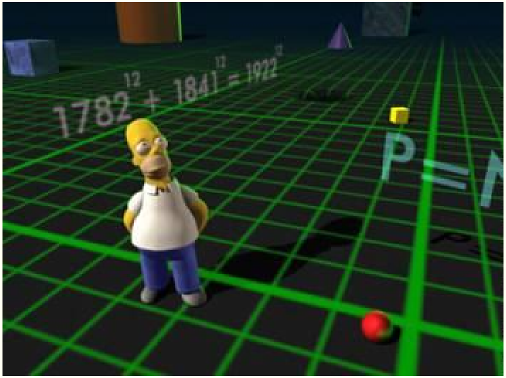
Preprocessing values

$$\begin{tikzcd}
 & & & & \text{Top32} \arrow[dl] \arrow[drr] & & & \\
 & & & \text{Ptr32} \arrow[dlll] \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] & & & \text{Num32} \arrow[dl] \arrow[dr] & \\
\text{DPtr32} \arrow[drrrr] & \text{NPtr32} \arrow[drrr] & \text{SPtr32} \arrow[drr] & \text{HPtr32} \arrow[dr] & \text{GPtr32} \arrow[d] & \text{Num32B0} \arrow[dl] & .. & \text{Num32B32} \arrow[dlll] \\
 & & & & \text{Bot32} & & &
\end{tikzcd}$$

Assertion:

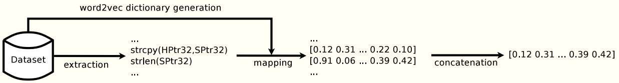
Machine Learning algorithms cannot deals with values like string, pointers, integers, that why replace them with meaningful labels.

n-space embedding

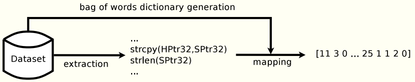


.. from traces to vectors ..

- Several options are available, we tried two:
  - Word2vec [1]



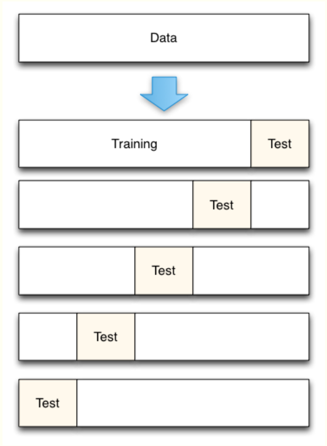
  - Bag of words



Once we convert every trace, we have everything ready to start doing some learning!

Model training/inference

Training and Testing



Formally,

1. Split the dataset into K equal partitions (or "folds").
2. Use fold 1 as the testing set and the union of the other folds as the training set.
3. Calculate testing accuracy.
4. Repeat steps 2 and 3 K times, using a different fold as the testing set each time.
5. Use the average testing accuracy as the estimate of out-of-sample accuracy.

ALERT!
You should **never** use the same data to **train and evaluate** in Machine Learning.

The usual suspects

- Logistic regression
- Random forests
- Multi-layer Neural Networks

Every classifier will be trained with different parameters and we will report the best test accuracy.

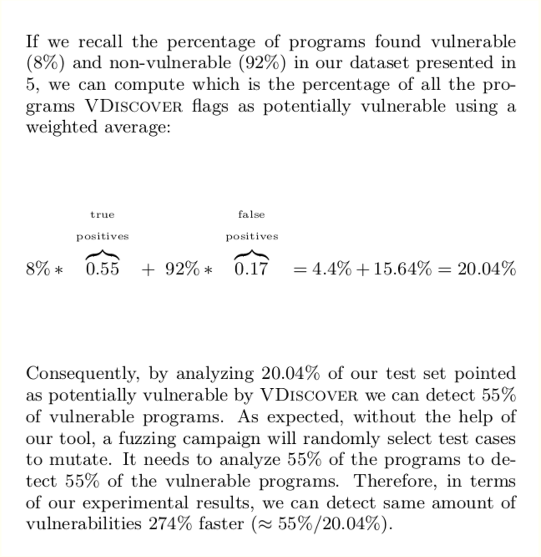
Testing on new data

Prediction accuracy

	Flagged	**Not Flagged**
Flagged	**55**%	17%
Not Flagged	45%	**83**%

These results are obtained using Random Forest (scikit-learn) in 1-3 grams dataset

Prediction accuracy



If we recall the percentage of programs found vulnerable (8%) and non-vulnerable (92%) in our dataset presented in 5, we can compute which is the percentage of all the programs VDiscover flags as potentially vulnerable using a weighted average:

$$8\% * \overbrace{0.55}^{\text{true positives}} + 92\% * \overbrace{0.17}^{\text{false positives}} = 4.4\% + 15.64\% = 20.04\%$$

Consequently, by analyzing 20.04% of our test set pointed as potentially vulnerable by VDiscover we can detect 55% of vulnerable programs. As expected, without the help of our tool, a fuzzing campaign will randomly select test cases to mutate. It needs to analyze 55% of the programs to detect 55% of the vulnerable programs. Therefore, in terms of our experimental results, we can detect same amount of vulnerabilities 274% faster ($\approx$ 55%/20.04%).

Give me a break!

Workshop Time!

Installing VDiscover

Make sure you install a recent version, not the ancient version from the Ubuntu repositories (you can download packages here)

1. Setup a VM:

```
vagrant init ubuntu/trusty32
vagrant up --provider virtualbox
vagrant ssh -- -X
```

2. Take some minutes to install basic stuff (git, python-setuptools, python-sklearn ..)

```
git clone https://github.com/CIFASIS/vdiscover-workshop
git clone https://github.com/CIFASIS/VDiscover
cd VDiscover
./setup.py install --user
```

(don't forget to append "PATH=$PATH:~/.local/bin" to your .bashrc)

VDiscover

- Open source (GPL3) and available here: `http://www.vdiscover.org/`
- Written in Python 2:
  - python-ptrace
  - scikit-learn (and dependencies)
- Composed by:
  - tcreator: test case creation
  - fextractor: feature extraction
  - vpredictor: trainer and predictor
  - vd: a high level script to save time extracting data
- Trace should be collected in x86 (because i'm lazy!)

Setting up a test case

```
$ printf '<b>Hello!'  > test.html
$ tcreator --name test-html --cmd "/usr/bin/html2text
file:$(pwd)/test.html" out
```

Workshop Time!

Experiment adding and removing arguments and files to check how test cases are created.

Setting up a test case

```
$ printf '<b>Hello!'  > test.html
$ tcreator --name test-html --cmd "/usr/bin/html2text
file:$(pwd)/test.html" out
```

Workshop Time!

Experiment adding and removing arguments and files to check how test cases are created.

Collecting my first trace (1)

```
$ fextractor --dynamic out/test-html/ > trace1.csv
$ cat trace1.csv
out/test-html/ strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32
strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32
strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32
strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32
strcmp:0=GxPtr32 strcmp:1=GxPtr32 ..
```

Workshop Time!

Take a few minutes to extract traces from other programs and how to include/exclude events from different modules (–inc-mods/–ign-mods)

Collecting my first trace (1)

```
$ fextractor --dynamic out/test-html/ > trace1.csv
$ cat trace1.csv
out/test-html/ strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32
strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32
strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32
strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32
strcmp:0=GxPtr32 strcmp:1=GxPtr32 ..
```

Workshop Time!

Take a few minutes to extract traces from other programs and how to include/exclude events from different modules (–inc-mods/–ign-mods)

Collecting my first trace (2)

```
$ printf '<baaa>Bye!'  > test.html
$ fextractor --dynamic out/test-html/ > trace2.csv
$ cat trace2.csv
out/test-html/ strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32
strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32
strcmp:0=GxPtr32 strcmp:1=GxPtr32 strcmp:0=GxPtr32
strcmp:1=GxPtr32 strcmp:0=GxPtr32 strcmp:1=GxPtr32
strcmp:0=GxPtr32 strcmp:1=GxPtr32 ..
```

It looks exactly the same!!

.. but in fact, they are not. Later, we are going to show how to **easily** visualize traces..

ZZUF dataset (1)

1. Collect binary programs from /usr/bin or /bin. For instance: "/usr/bin/identify"
2. Fetch different seed files from the fuzzing project (https://files.fuzzing-project.org/). For instance: "audio.wav"
3. Extract and select random command line arguments (manfuzzer). For instance: "-verbose -gamma 42"
4. Collect the dataset:
   - 4.1 Use zzuf to fuzz command line using every possible seed file.
   - 4.2 Detect crashes, aborts, timeouts and very large allocations.
   - 4.3 Record traces using VDiscover.
5. Train and test using VDiscover.

A detailed explanation of this dataset is available here:
`http://www.vdiscover.org/OS-fuzzing.html`

ZZUF dataset (2)

- cmds.csv.gz: 64k command-line to fuzz
- traces.csv.gz: sampled and balanced traces ready to be trained and tested
- zzuf.csv.gz: output from zzuf after fuzzing

To split the data in train and test sets:

```
$ ./split.py dataset/traces.csv.gz 42
```

Training and testing a bug predictor

- Training:
  ```
  $ vpredictor --dynamic --train-rf data/42/train.csv --out-file
  model.pklz
  ```
- Testing:
  ```
  $ vpredictor --test --dynamic --model model.pklz data/42/test.csv
  --out-file predicted.out
  ...
  Accuracy per class:  0.7 0.85
  Average accuracy:  0.77
  ```

Visualizing test cases

- Collecting data:
  ```
  $ tar -xf files.fuzzing-project.org.tar.gz
  $ vd -i seeds/files.fuzzing-project.org "/usr/bin/file @@" -o
  file-traces.csv
  ```
- Clustering using bag of words (bow) and display:
  ```
  $ vpredictor --cluster-bow --dynamic file-traces.csv
  ```

Question

- How test cases are clustered?

Extended exercises

1. Add a command line flag to VDiscover to suppress the hooking of certain event.
2. Visualize some new test cases with pnginfo:
   2.1 Collect traces and plot test cases using the png test suite (PngSuite-2013jan13.tgz) with pnginfo
   2.2 Observe the results and explain how the test cases are clustered.
3. Limitations in VDiscover implementation:
   3.1 Identify issues in the trace collection using the seed from the fuzzing-project with html2text.
   3.2 Suggest a possible solution.
4. Discuss:
   4.1 How the visualization can be used in vulnerability discovery?

References

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean.
Efficient estimation of word representations in vector space.
CoRR, abs/1301.3781, 2013.

Mayhem Team.
List of mayhem bugs.
`https://bugs.debian.org/cgi-bin/pkgreport.cgi?submitter=alexandre%40cmu.edu`, 2013.

Mayhem Team.
Reporting 1.2K crashes.
`https://lists.debian.org/debian-devel/2013/06/msg00720.html`, 2013.

Vagrant.
Vagrant: Development environments made easy.
`http://www.vagrantup.com/`, 2014.

Fabian Yamaguchi, Felix Lindner, and Konrad Rieck.
Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning.
In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11. USENIX Association, 2011.