IDENTIFYING AND CLUSTERING ATTACK-DRIVEN CRASH REPORTS USING MACHINE LEARNING

A Dissertation
Presented to
The Academic Faculty

By

Ibtehaj M. Alzahrani

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Computer Science

Georgia Institute of Technology

May 2019

IDENTIFYING AND CLUSTERING ATTACK-DRIVEN CRASH REPORTS USING MACHINE LEARNING

Approved by:

Prof. Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Prof. Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Dr. Taesoo Kim
School of Computer Science
Georgia Institute of Technology

Date Approved: April 25, 2019

ACKNOWLEDGEMENTS

I would like to thank my advisor Prof. Wenke Lee for his guidance, encouragement, and resources he provided me with. I would like to thank Dr. Hong Hu, Chenxiong Qian for sharing their knowledge and insights and assisting me in finding the direction for my research. My sincere thanks to Ren for his patience to address technical problems and share the dataset used in his system. I would like to thank Carter Yagemann for his valuable feedback. I would like to thank Prof. Mustaque Ahamad and Dr. Taesoo Kim for serving as my committee members. I gratefully appreciate and acknowledge the funding received towards my Master's degree from the Kingdom of Saudi Arabia scholarship.

I am grateful to my husband Abdullah who has been my most tremendous support and my strongest motivation. I am also grateful to my parents, Reda and Mohammad whose love, support and prays brought me thus far.

Finally, my greatest thanks are to my children, Ahmed, and Dema who their smiles make me the happiest woman in the world. You have given me the strength and courage to keep going, and without you, I would not be where I am today. I LOVE you unconditionally. I dedicate my thesis to my parents, husband, and children.

TABLE OF CONTENTS

Acknowledgments iii

List of Tables vi

List of Figures viii

Chapter 1: Introduction and Background 1

Chapter 2: Design Overview 2

2.1 Classification Phase 3

2.1.1 General Features 3

2.1.2 Heap Classifier 6

2.1.3 Shellcode Classifier 9

2.1.4 Fmtstr Classifier 10

2.1.5 ROP Classifier 12

2.1.6 Data Standardization 14

2.2 Clustering Phase 15

Chapter 3: Experiments and Results 17

3.1 Evaluation Metrics 18

3.2 Selecting Machine Learning Algorithm 19

3.3 Classifiers Evaluation 21

3.3.1 Features Importance 23

3.4 System Evaluation 23

3.5 System Performance 27

Chapter 4: Conclusion 28

References 29

LIST OF TABLES

2.1 Core Signals on x86 Architectures 6

2.2 Summary of the feature set for each classifier 7

2.3 One-Hot Encoding for Format String Payload 11

2.4 Clustering Features 15

3.1 Summary of the training and validation sets. There are in total 7375 crash reports. 100 crash reports from ffmpeg contains three attack types and 100 of them contains two attack types, thus, the bottom right corner have an extra 300. The columns show the number of crash reports for each binary per behavior whereas the raws show the number of crash reports per malicious behavior. 17

3.2 Summary of the test set. There are in total 1843 crash reports. The columns show the number of crash reports for each binary per behavior whereas the raws show the number of crash reports per malicious behavior. 18

3.3 Comparison of statistical Classifiers - Heap Classifier 20

3.4 Comparison of statistical Classifiers - Shellcode Classifier 20

3.5 Comparison of statistical Classifiers - Fmtstr Classifier 20

3.6 Comparison of statistical Classifiers - ROP Classifier 20

3.7 Summary of the accuracy score of the training, validation, and testing sets for all the Classifiers on the Classification phase. 21

3.8 Summary of the misclassified crash reports on the training, validation, and testing sets. 22

3.9 Summary of the clustering performance on *ffmpeg* and *php* crash reports. . . 25

3.10 Summary of the clustering results for *tachikoma* crash reports. 26

LIST OF FIGURES

2.1 System architecture. It classifies potential malicious crash reports caused by four attack types, and cluster them based on the exploited vulnerabilities. 2

2.2 Validating EBP 5

2.3 Senario on Considering Gadgets 13

3.1 ROC curves for Random Forest Classifiers - Heap, Shellcode, Fmtstr and ROP Classifiers 23

3.2 Featuers Importance - Shellcode Classifier 24

3.3 Featuers Importance - Fmtstr Classifier 24

3.4 Featuers Importance - ROP Classifier 24

3.5 Featuers Importance - Heap Classifier 24

3.6 ROP Classifier - when removing two features: one feature at a time. 25

SUMMARY

We propose a tool to identify crashes caused by filed exploits from benign crashes, and cluster them based on the exploited vulnerabilities to prioritize crashes from a security point of view. The tool extracts features from crash reports and decides whether a crash caused by malicious behavior or not. In the case of malicious behavior, it identifies the attack type that generates the crash report; we are focusing on four attack types which are Heap exploitation, Shellcode injection, Format String attack, and Return Oriented Programming. Further, it clusters the crash reports based on the exploited vulnerabilities.

CHAPTER 1
INTRODUCTION AND BACKGROUND

Although softwares go into severl steps to ensure its overall reliability and dependability, a bug can be existed after production and crash reporting systems play an important role in identifying and debugging crashes in software systems[1]. Crash reporting systems are common these days in most widely deployed software systems [1]. When a fault happens in a process, the system terminates the process and generates a detailed problem report. Vendors collects such reports automatically and classifies them to minimize programmer effort [2]. Crashes generated because of a fault in which it can be benign casued by, not limited to but including, incorrect code or improper interaction with third party applications [1] or malicious. Classifing and prioritizing crashes should be based on the level of risk of the bugs in such a way that a crach generated becasue of a bug that is targeted by attackers is much more important to be fixed than a crash generated by benign reasons, hence, the reliability and dependability of the system increased. Crash reporting files contain valuable information that can be used to identify not only the root cause of a crash but as well whether it is benign or malicious.

We propose a tool that can automatically identify crashes generated by attackers from benign ones, and then cluster the attacker-driven crashes based on the root cause. An malicious crash can be identified as one or more of the most popular and widely found on crashes caused by attacks which are heap exploitation, Retuned-Object Programming attacks, shellcode injection, and format string attack. The tool is composed of feature extraction, classification, and clustering phases in which the first phase extracts features for machine learning models that used by the tool, the second phase classifies crashes based on the types of attacks that are found, and the third phase clusters the cores based on the exploited vulnerabilities.

CHAPTER 2

DESIGN OVERVIEW

In this chapter, we present the overall design of the system, and further, we discuss the features that are used by the classification and clustering phases. The system first analyzes crash reports and extract features from them for both the classification and clustering phases. Second, the data is fed to the classification classifiers to identify whether a crash report is benign or malicious. We identify a crash report as malicious if it has one or more of the following threats: heap exploitation, Retuned-Object Programming attacks, shellcode injection, and format string attack. We are focusing on these attacks as they are the most common attacks. We identify crash reports as benign if the results from all the four classifiers are negative, zero. Third, compose the clustering dataset from the clustering feature set which extracted during the feature extraction phase and based on the results from the classification process. Finally, we feed the composed dataset to the clustering algorithm to cluster the crash reports based on the exploited vulnerabilities. Figure 2.1 shows the overall architecture.

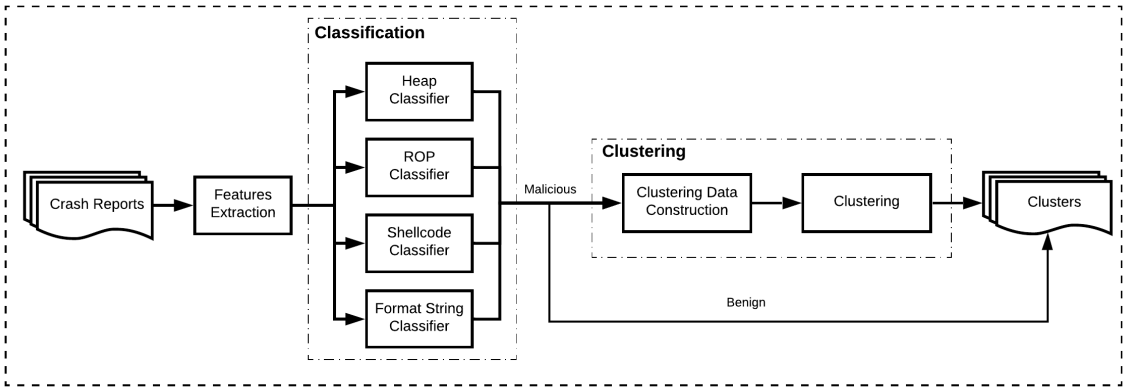


Figure 2.1: System architecture. It classifies potential malicious crash reports caused by four attack types, and cluster them based on the exploited vulnerabilities.

2.1 Classification Phase

During the classification phase, the crash is labeled either benign or malicious. A crash report can have one or more of the threats we examined the crash reports against them. There are four different classifiers: (1) Heap classifier which detects if a crash report caused by heap exploitation, (2) ROP classifier which detects whether the ROP attack exists on a crash report, (3) Shellcode classifier which recognizes when a crash report has a shellcode injection, and (4) Format String (Fmtstr) which identifies if a program is vulnerable to format string attacks or not. All the classifiers are binary-based classifiers in which each core labeled as either 1 or 0, where one means that the attack type that the classifier assigned for is found on the crash report and 0 indicates it not found. During the feature extraction phase, a set of features are extracted to be used by Heap, ROP, Shellcode and Fmtstr classifiers. In this section, we present a clear overview of the feature set of each classifier and the purpose behind choosing them. We first discuss a set of features that are used by more than one classifier which we named them general features. General features are the features that have general information regarding the overall of the crash report and are not for a specific attack.

2.1.1 General Features

We believe that general features provide a pattern for every attack. Primarily, when a crash happens in the middle of the exploitation payload.

Stack Pointer Register Validation

The Stack Pointer register indicates the location of the last element used on the stack. We validate the value stored on the Stack Pointer register on a crash report by examining whether the value belongs to the Stack memory region or not. The adversary can use Stack pivot technique to control the victim program. The adversary can launch the stack pivoting

by placing the payload into another memory region and manipulating the stack pointer to point to that region. After lunching Stack pivoting technique, the adversary moved the control from the actual Stack to fake Stack; thus, the Stack Pointer Register is corrupted. Validating Stack Pointer register helps in distinguishing ROP attack that uses Stack Pivoting method.

Stack Base Pointer Register Validation

Stack Base Pointer register points to the top of the stack when the function is first called. After the function is executed, it points to the base of the Stack. On Stack-based buffer overflow, the Stack Base Pointer register is most likely to be overwritten. Therefore, validating the Stack Base Pointer register is a good indicator of malicious behavior. We validate Stack Base Pointer register by first checking whether the value stored on the register belongs to the Stack memory region or not. When a function is called, a function prologue is executed in which the current value of Stack Base Pointer register pushed into the Stack to save its current value. Then, the value of Stack Pointer register is moved to Stack Base Pointer register. Finally, a space for the frame is assigned by subtracting n words from Stack Pointer register. Therefore, iterating over the values stored on the Stack Base Pointer register gives us the previous Stack frames base which further points to the Stack. Hence, we keep checking whether the values inside the pointers are pointing to the Stack til we reach zero as the entry point usually sets up the initial value of the Stack Base Pointer Register to zero. Reaching zero indicates that we reach the end of the call Stack which means that the register is probably not corrupted. Figure 2.2 clarifies the process of validating Stack Base Pointer Register.

Instruction Pointer Register Validation

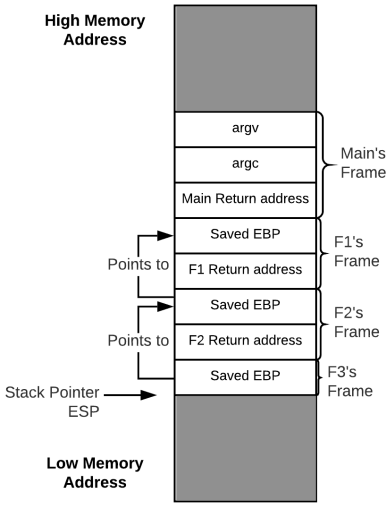
Instruction Pointer register points to the next Instruction to be executed. In the case of Shellcode injection, the Instruction Pointer register is corrupted in which it points another

```c
#include <stdio.h>

void F3(){
    printf("Hello, World!");
}
void F2(){
    F3();
}
void F1(){
    F2();
}

int main( int argc, char *argv[] ){
    F1();
    return 0;
}
```

(a) A Simple C Program



(b) A call stack

Figure 2.2: Validating EBP

memory region where the Shellcode injected. On Heap-based vulnerabilities, the adversary can use Shellcode injection, ROP chain, or both Shellcode injection and ROP to launch a complete desired exploitation. Therefore, we believe that including the validation of the Instruction Pointer register as a feature for Shellcode and Heap classifiers is a significant indicator of malicious crashes.

Signal

A program crash has a specific crash signal in which it tells the reason behind the crash - a Unix core signal shown in Table 2.1 [3], where the first column is the number of the crash signal and the second column is the signal name. Usually, crash reports crashed with a signal number 11. However, when an injected shellcode payload is inaccurate. Then, when the Instruction Pointer Register wants to jump to the next instruction that is invalid instruction, the program crashes and leave behind a crash report with signal number 4 which indicates that the next Instruction to be executed is invalid. Additionally, it is not likely for a crash report that has signal number 8 to be caused by any of the threats we are focusing on them. Therefore, we believe that having the crash signal as a feature for Shellcode and

Table 2.1: Core Signals on x86 Architectures

Core Signal Number	Comment
3	Quit from keyboard
4	Illegal Instruction
5	Trace/Breakpoint Trap
6	Abort
7	Bus error -Bad memory access
8	Floating-point exception
11	Segmentation Fault - Invalid memory reference
24	CPU time limit exceeded
25	File size limit exceeded
31	Bad system call

ROP classifiers gives us a clue on whether a crash report is malicious or not.

2.1.2 Heap Classifier

In addition to the Instruction Pointer Register Validation from general features, we have introduced a set of features for the Heap Classifier. The features is added in which it gives the classifier much information regarding the possibility of heap exploitation existing. To extract the features for Heap classifier, we first have to reconstruct the heap memory area by calculating the address of the main arena using the main arena offset that is searched for on the debug version of Libc. After locating the main arena address, we then locate the top chunk and the Heap start and end addresses. The feature set for Heap Classifier summarized in Table 2.2.

Number of Violation Found on Top Chunk

We validate the top chunk on two aspects. First, we verify whether the heap region is valid in which it is not executable and have correct boundaries. Second, we validate the top chunk address by inspecting whether the top chunk size equals to the maximum Heap size which usually assigned to 0x21000.

$$MaxTopChunkSize = HeapEndAddress - HeapStartAddress \tag{2.1}$$

Table 2.2: Summary of the feature set for each classifier

Classifier	Features
Heap Classifier	Is Instruction Pointer Register Corrupted?
	Number of Violation Found on Top Chunk
	Number of Invalid Pointers
	Number of Chunks with Invalid in_use bit
	Number of Chunks with Invalid prev_size
	Number of Misaligned Chunks
	Number of Chunks on Fast Bin with Invalid size
Shellcode Classifier	Signal
	Is Instruction Pointer Register Corrupted
	Number of Jump Instruction
	Number of System Call Instruction
	Number of Call Instruction
	Number of Move Instruction
	Number of Xor Instruction
	Number of Nop Instruction
Fmtstr Classifier	Is Stack Base Pointer Register Corrupted?
	Are Format String vulnerable Functions exist?
	%
	$
	n
	h
	u
	d
	o
	p
	s
	f
	x
	c
	Number of Format String Symbols
	Number of Format String Possible Payloads
	Minimum Gap
ROP Classifier	Signal
	Is Stack Pointer Register Corrupted?
	Is Stack Base Pointer Register Corrupted?
	Number of Unexpected Instruction Pointers On Stack
	Number of Gadgets on Data
	Number of Code Blocks
	Number of Heap Chunks Invalid Pointers

Third, we validate the top chunk's in_use bit which it has to be not set because the top chunk's previous chunk should be allocated, malloc'ed. We believe that any violation on the top chunk's in_use bit and size as well as its position is a sign of malicious activities.

Number of Invalid Pointers

It is crucial that we keep track on freed chunks pointers because the adversary can use the opportunity to manipulate the forward pointer, FD, or/and backward pointer, BK. FD pointers exist on freed chunks that kept on bins that maintain chunks with either single and doubly linked list. BK pointers exist only on the bins that maintain chunks with a doubly linked list. The malloc implementation uses these pointers to keep track of freed chunks. The adversary can manipulate the pointers to control program execution. Therefore, including the number of invalid freed chunks pointers is an essential indicator of malicious behaviors.

Number of Chunks with Invalid in_use bit

We verify whether the in_use bit of a chunk sets correctly or not. We investigate all chunks including freed and allocated chunks. The adversary can change the in_use bit during forged chunk creation process as well as overwriting another chunk. Consequently, counting the number of the chunks with invalid in_use bit adds a piece of evidence whether the crash report is malicious.

Number of Chunks with Invalid prev_size

In malloc implementation, when a chunk is freed, the chunk placed after it set the prev_size field to the size of the freed chunk. The prev_size field is used only by the implementation when the previous chunk is freed. Therefore, when a chunk is allocated, the chunk after has to have zero on the prev_size field. It is common that the adversary manipulates previous size field of a chunk stored. Therefore, counting the number of chunks that have invalid

prev_size helps indicates whether a crash report is malicious.

Number of Misaligned Chunks

We further validate whether a chunk is misaligned with other chunks including top chunk or whether the chunk lies off Heap region's boundaries. Ideally, chunks do not overlap with each other or lie out of its region, Heap; however, it is obtainable by the adversary to mistakenly overwrite a chunk size with an incorrect size which results on such signs. Therefore, the Number of misaligned chunks feature is a remarkable indicator of whether heap exploitation exists.

Number of Chunks Located on Fastbin with Invalid Size

All chunks stored on one fastbin, have a fixed size. We validate whether the chunk size equals to the expected chunk size for a fastbin or not. Therefore, counting the number of chunks located on fastbin and having a wrong size is an indicator of unauthorized activities.

2.1.3 Shellcode Classifier

In addition to Instruction Pointer register validation from the general features, we feed the classifier with the following features.

Number of Jump, System Call, Call, Move, Xor and Nop Instructions

We scan writable and executable memory regions for any possible shellcode. A disassembler tool would disassemble any bytes to opcode even if they are not actual opcodes. Therefore, instead of counting the instructions found on writable, executable memory regions, we count the frequency of most commonly used instructions. We count the frequency of *jump*, *system call*, *call*, *move*, *xor*, and *nop* instructions. We believe that in order for the adversary to control program execution, the adversary requires at least one of the following instructions: *jump*, *system call*, *call*, *move* and *xor*. The adversary would require to clear

the registers for the exploit to work. Hence, the adversary would want to use *xor*. Similarly, it is required to execute fully functional shellcode to use jump and move instructions. Additionally, in order for the adversary to invoke an Operating System function, the adversary requires to use *system call* instruction. We further count the frequency of *nop* instruction as attackers tend to use *nop* sled in order to have much reliable payload. As a result, we believe that it is adequate to have the frequency of *nop* instruction because the higher the number of *nop* instruction is the higher probability of having shellcode injection on the crash report.

2.1.4 Fmtstr Classifier

The Fmtstr Classifier has a set of features that we believe are important to detect Format String attacks. The Classifier has only the validation of Stack Base Pointer register from general features as a feature because it is possible that the adversary overwrites the register.

Format String vulnerable Functions

We check whether Format String unsafe functions exist on the binary or not because it is not possible to have format string attack when the functions do not exist. However, when they exist, it is possible to have a Format String attack but not necessary. Therefore, it is necessary that we feed the Classifier with the information about the existence of the vulnerable functions because it gives us a clue whether the attack is possible or not.

Encoded Format String Payload

We are applying patterning match on all the writable memory regions including Heap, and all the found payloads are encoded using one-hot encoding technique — one-hot encoding is widely used on machine learning for encoding string features. We create one feature for each possible symbol that can be used on format string payloads. For instance, the feature % represents the exiting of the percentage sign symbol on writable memory regions. When

Table 2.3: One-Hot Encoding for Format String Payload

Feature	Range of Values
%	0 or 1
$	0 or 1
n	0 or 1
h	0 or 1
u	0 or 1
d	0 or 1
o	0 or 1
p	0 or 1
s	0 or 1
f	0 or 1
x	0 or 1
c	0 or 1

% exits, the feature has a value of one and when it does not exist, it has a value of zero. Encoding possible payload results in a total of 12 features for each symbol described on Tabel 2.3.

Number of Symbols

It is not sufficient for the classifier to tell whether Fmtstr attack exists by knowing only the symbols. Therefore, we include the number of symbols found in writable memory regions as a feature. The number of symbols gives us a clue on whether format string possible.

Number of Format String Possible Payloads

As our tool is implemented to be used by any binary, we considered the case where the binaries are storing files including images which is the case on FFmpeg. To avoid misclassification with encoded files, we introduce a feature that counts the number of payloads on writable memory regions. We believe that the number of possible payloads give us a clue on whether the attack exists or not.

Minimum Gap

In addition to the number of Format String possible payloads, we introduce a feature that specifies the minimum gap between all possible payloads. We first calculate the gaps between all payloads and then, take the minimum gap. Gap value indicates whether the

exploit is feasible or not.

2.1.5 ROP Classifier

The ROP Classifier has the validation of Stack Pointer, and Stack Base Pointer register features from the general features. Besides, it has a set of feature discussed in this section.

Number of Unexpected Instruction Pointer

It is practically complex to detect ROP chain because the ROP chain has the same structure of regular program execution. Therefore, instead of looking for ROP chain, we are looking for what characterized ROP chain which is Gadgets. Gadgets distinguish ROP attack from other attacks, and a successful ROP chain contains at least one or more gadgets. Consequently, we count the number of possible gadgets on call Stack. We scan byte by byte in order to intercept misaligned pointers. A wrong assumption of the size of the payload yields to have the Gadget pointer to be misaligned. For each pointer located on the area we examined on the Stack, we check whether the pointer points to the *.text* section of the binary or any library of the included libraries. We exclude functions addresses which can be used by the program in case of *call* instructions, and we further exclude any return addresses. We detect return addresses by examining whether the previous instruction which is located just before the instruction that the possible return address points to, is a call instruction or not. If previous instruction is call instruction, we assume that the pointer is a return address. We consider the pointer to be a Gadget pointer by verifying whether a return instruction exists on the instruction or the following instructions. Figure 2.3 describes the scenario in which we consider pointers as a possible gadget.

Number of possible Gadgets on Data

When the Stack Pointer register is corrupted and pointing to a writable memory region, it is a very high chance caused by Stack Pivoting technique. Therefore, we interduce a feature

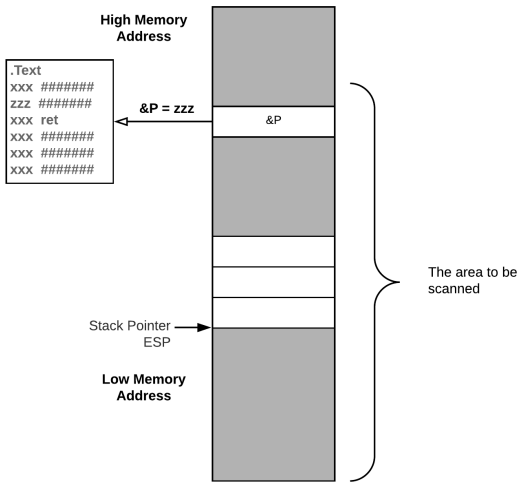


Figure 2.3: Senario on Considering Gadgets

that counts the number of possible the gadgets on only writable memory regions and not on the Stack. We expect that caused by benign reasons to not have pointers on *data* points to an executable memory region. Therefore, we believe that having this feature is a significant identification of the existing of Stack Pivoting.

Number of Code Blocks

We include the number of code blocks in which we count the number of code blocks found on writable memory regions by detecting instructions that create code blocks such as Call instruction. The registers are most likely to be corrupted in case of Shellcode injection; therefore, it would yield to misclassifying a crash report has Shellcode injection as having ROP attack. Therefore, we include the number of code blocks as a feature to reduce the possible false positive. The false positive in this case is not imperfect because, in the end, it is a malicious crash report; however, it is essential to be as precise as possible in order to have a highly accurate clustering results on the clustering phase because the clustering phase is mainly depending on the accuracy of the classification phase.

Number of Heap Chunks Invalid Pointers

Similarly, we include the number of invalid pointers for heap chunks because Heap-based control hijacking could manipulate the registers; thus, we include the number of manipulated pointers as a feature in order to avoid the False positives of the misclassifying of crash reports caused by Heap exploitation as caused by ROP.

2.1.6 Data Standardization

After constructing the dataset and before feeding the data to the Classifiers, we standardized the data by using the standard score, z-score. We apply the z-score equation, shown in Equation 2.1, to each data point in the dataset. The mean and the standard deviation of each feature is calculated independently from each other. Important to note that we did not measure the mean and the standard deviation on the entire dataset; training, validation and testing sets. Instead, we calculated them based on the training set only because we aim to generalize the models. If the mean and the standard deviation of the validation and test sets are knowing to the classifiers during training step, then, we cannot say that the accuracy of the classifier is accurate because in this case, the classifiers know the mean and the standard deviation of the validation and testing set in advances. Therefore, we did not get the mean and standard deviation of the validation and test sets, and we standardized them using the mean and the standard deviation calculated from the training set in which we guarantee the correctness of our Classifiers. Equation 2.2 is the z-score equation that we apply on the training set.

$$z = \frac{x - \mu}{\sigma} \tag{2.2}$$

Where z is the z-score of a data point, x is the data point, μ is the mean, and σ is the standard deviation.

Table 2.4: Clustering Features

Feature	Range of Values
Heap Start Offset	Integer, ≥ 0, -1 when no payload found on Heap
Stack Start Offset	Integer, ≥ 0, -1 when no payload found on Stack
Data Start Offset	Integer, ≥ 0, -1 when there is payload found on Heap or Stack, or no payload found on Data

2.2 Clustering Phase

Many crash reports are caused by the same bug [4]. Therefore, we cluster crash reports based on the exploited vulnerabilities in order to reduce the time developers take to debug them. We first cluster them to malicious and benign in which we prioritize them based on the severity of the bug from a security perspective. We further cluster the malicious crashes based on the exploited vulnerabilities. Clustering malicious crashes is not a simple task as in most malicious crashes; registers are manipulated and corrupted purposely. As a result, it is useless to cluster the crashes based on call stack. For instance, in case of stack pivot, the Stack Pointer register is no longer pointing to the actual Stack memory; instead, it points to another memory region. Moreover, in Stack buffer overflow vulnerabilities, Stack Base Pointer register is overwritten which makes it infeasible to get the call stack. Therefore, it is hard to identify the root cause based on the call stack because each crash report can have a different call stack based on the exploits. Further, clustering crash reports based on the Instruction Pointer register is insufficient because when the adversary controls the program execution, the register can be pointing to another memory region. The ineffectiveness of these methods proved in [5]. Moreover, it is not sufficient to cluster crash reports based on signatures taken from exploits because different attackers can exploit a bug differently. Therefore, we cluster the crashes based on the offset of the exploit.

During the feature extraction phase, we still do not know whether a crash report is malicious or benign and which attacks existed within the malicious crash reports. Additionally, a multi-stage exploit involves more than one attack type and exist in different memory re-

gions. As a result, we are collecting offsets for all the possible exploitation payloads on all the memory regions. Then, we confirm which attack is existed by feeding the data to the Classifiers from Classification phase. In other words, the proper offsets will be selected based on the results from the Classification phase. For instance, for an exploit placed on Heap, the smallest offset considered the start offset of the payload. We calculate the offset differently for exploits exists on the Stack. As the memory growth towards lower memory addresses on the Stack and the environment variables are placed on top of the stack, we set the Stack base after the environment variables in order to have a more stable offset. The offset for any exploit exists on the stack are calculated by subtracting the payload start address from stable Stack base. Three possible memory regions can hold exploitation: Heap, Stack, and Data. We only consider the offset for exploit found on Data, if there is no offset found on Heap or Stack. Using the mentioned details the tool automatically constructs the data to feed to the clustering algorithm. The final clustering features summarized in Table 2.4.

Among all the existing clustering algorithm we selected Mean Shift algorithm because it does not require specifying the number of clusters and we can control the result by one parameter, the bandwidth [6]. Mean shift utilize the density of the points to generate a reasonable number of clusters.

CHAPTER 3

EXPERIMENTS AND RESULTS

We collected 14348 crash reports from the data that is used by CORONER [5]. We divided our dataset into two sets: (1) a set that is used to train, validate and test the Classifiers holding 9218 crash reports, and (2) a set that is used to evaluate the entire system containing 5130 crash reports. We further split the first set into three sets: (1) a training set containing 60% of the 9218, (2) a validation set containing 20% of the 9218, and (3) a testing sets containing 20% of the 9218. This policy is applied to all four classifiers. Table 3.1 described the training and validation sets. The selection of the validation set is selected randomly; however, we carefully selected the test set in which it contains crash reports from all possible categories: Benign, Heap, Shellcode, Fmtstr, and ROP. The Classifiers never see the crash reports on the test set and 97.3% of the attacker-driven crash reports are generated by binaries that their crash report never been seen by the Clssifiers. In addition to attacker-driven crash reports, we included crash reports that are caused benignly selected randomly from ffmpeg and php. In total, we have crash reports on the test set generated by six new binaries preserved for the test set and ffmpeg and php. The test set is selected in which we use a unique data, generated by unseen binaries, as well as use all the available data. We

Table 3.1: Summary of the training and validation sets. There are in total 7375 crash reports. 100 crash reports from ffmpeg contains three attack types and 100 of them contains two attack types, thus, the bottom right corner have an extra 300. The columns show the number of crash reports for each binary per behavior whereas the raws show the number of crash reports per malicious behavior.

Identified Behavior	Ffmpeg	PHP	Students	sum
Benign	1397	1075	0	2472
Heap	300	0	251	551
Shellcode	100	0	587	687
Fmtstr	0	200	1575	1775
ROP	200	0	1990	2190
sum	1997	1275	4403	7675

Table 3.2: Summary of the test set. There are in total 1843 crash reports. The columns show the number of crash reports for each binary per behavior whereas the raws show the number of crash reports per malicious behavior.

Identified Behavior	Ffmpeg	PHP	Students	sum
Benign	269	238	0	507
Heap	50	0	236	286
Shellcode	0	0	365	365
Fmtstr	0	0	300	300
ROP	0	0	385	385
sum	319	238	1286	1843

also consider taking an equal portion of each of the four exploitation type — the test set with their associated behavior highlighted in Table 3.2. Our experiments and evaluation of the tool completed on a 3.1GHz dual-core Intel Core i5 running macOS 10.13.6.

3.1 Evaluation Metrics

In here we present an explanation of the metrics that we used to evaluate and assess different Machine Learning Classifiers. We are using the accuracy score to evaluate how well the model performs. The accuracy score is the percentage of correctly classified instances by the classifiers.

$$Accuracy = \frac{TruePositive + TrueNegative}{NumberofInstances} \tag{3.1}$$

Where is the True Positive is the number of positive instances on a set that the classifier identified them as positive, and the True Negative is the number of negative instances that the classifier correctly identified them as negative. The Recall score is the same as the true positive rate and Precision is the True Positive rate over the number of instances that are predicted by the classifier to be positive.

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{3.2}$$

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{3.3}$$

We further used F-measure to evaluate the Classification models and our clustering algorithm. F-measure combines recall and Precision. The best score for F-measure is one and the worst is zero.

$$F - measure = 2 \times \frac{recall \times precision}{recall + precision} \tag{3.4}$$

Besides F-measure, we used two additional metrics to evaluate the clustering algorithm, Homogeneity, and Silhouette coefficients. Homogeneity matrix is satisfied when all instances on a cluster belong to the same class. Homogeneity matrix returns a score in between zero and one where one is for perfectly homogeneous labeling and zero for non-perfect labeling. Silhouette coefficients are to measure the consistency of an instance to its cluster compared to the neighbor cluster. Computing Silhouette score depends the average distance between a data point and other data points on the same cluster and the average distance between the data point and other data points belongs to the neighbor cluster.

$$Silhouette = \frac{b - a}{max(a, b)} \tag{3.5}$$

Where a for a data point is the average of intra-cluster distance, and b is the average of nearest-cluster distance [7].

3.2 Selecting Machine Learning Algorithm

We performed experiments on 7375 crash reports, the training and validation sets, using the scikit-learn library [8]. We used 10-fold cross-validation to assess the effectiveness of several learning algorithms for our dataset. We compared a set of Machine Learning classifiers on the average score of cross-validation, the accuracy of the validation set, the true positive rate of the validation set, F-measure, and the time it takes to build the mode. Based

Table 3.3: Comparison of statistical Classifiers - Heap Classifier

Classifier	Average CV Accuracy	Validation Accuracy	Validation TP Rate	% F-measure	Model Building Time (ms)
Logistic Regression	96.8%	97.22%	97%	0.97	0.021
Decision Tree Classifier	98.46%	98.58%	99%	0.99	0.049
Linear Discriminant Analysis	95.86%	95.59%	96%	0.96	0.018
k-nearest Neighbors	97.98%	97.9%	98%	0.98	0.043
Gaussian Naive Bayes	77.29%	76.95%	77%	0.82	0.030
Support Vector Machine	97.44%	97.97%	98%	0.98	0.043
Random Forest Classifier	98.47%	98.58%	99%	0.99	0.086

Table 3.4: Comparison of statistical Classifiers - Shellcode Classifier

Classifier	Average CV Accuracy	Validation Accuracy	Validation TP Rate	% F-measure	Model Building Time (ms)
Logistic Regression	99.98%	100%	100%	1	0.019
Decision Tree Classifier	99.98%	100%	100%	1	0.035
Linear Discriminant Analysis	99.88%	99.86%	100%	1	0.021
k-nearest Neighbors	99.97%	100%	100%	1	0.048
Gaussian Naive Bayes	98.9%	98.98%	99%	0.99	0.027
Support Vector Machine	99.98%	100%	100%	1	0.037
Random Forest Classifier	100%	100%	100%	1	0.058

Table 3.5: Comparison of statistical Classifiers - Fmtstr Classifier

Classifier	Average CV Accuracy	Validation Accuracy	Validation TP Rate	% F-measure	Model Building Time (ms)
Logistic Regression	98.83%	98.44%	98%	0.98	0.014
Decision Tree Classifier	99.07%	98.44%	98%	0.98	0.016
Linear Discriminant Analysis	96.93%	96.54%	97%	0.97	0.013
k-nearest Neighbors	98.94%	98.37%	98%	0.98	0.026
Gaussian Naive Bayes	99.07%	98.51%	99%	0.98	0.022
Support Vector Machine	99.03%	98.51%	99%	0.98	0.055
Random Forest Classifier	99.06%	98.44%	98%	0.98	0.051

Table 3.6: Comparison of statistical Classifiers - ROP Classifier

Classifier	Average CV Accuracy	Validation Accuracy	Validation TP Rate	% F-measure	Model Building Time (ms)
Logistic Regression	95.50%	95.25%	95%	0.95	0.021
Decision Tree Classifier	99.78%	99.66%	100%	1	0.031
Linear Discriminant Analysis	95.88%	95.53%	96%	0.95	0.02
k-nearest Neighbors	99.69%	99.59%	100%	1	0.044
Gaussian Naive Bayes	80.71%	79.39%	79%	0.76	0.023
Support Vector Machine	99.19%	98.32%	99%	0.99	0.044
Random Forest Classifier	99.78%	99.67%	100%	1	0.099

Table 3.7: Summary of the accuracy score of the training, validation, and testing sets for all the Classifiers on the Classification phase.

Classifier	Training Set		Validation Set		Test Set	
	F-measure	Accuracy	F-measure	Accuracy	F-measure	Accuracy
Heap Classifier	0.98	99%	0.98	98%	0.98	98%
Shellcode Classifier	1	100%	1	100%	1	100%
Fmtstr Classifier	0.99	99%	0.99	99%	1	100%
ROP Classifier	1	100%	1	100%	1	100%

on the results presented in Table 2.3 to Table 2.6, we selected Random Forest algorithm to build our four classifiers: Heap, Shellcode, Fmtstr, and ROP classifiers. Random Forest algorithm gives the most accurate results among all of the algorithms for Heap and Shellcode Classifiers. For ROP classifier Although Decision Tree algorithm results are close to Random Forest Classifier, we selected Random Forest Classifier over Decision Tree Classifier, Decision Tree and Random Forest algorithms give the same accuracy which is the best accuracy. Decision Tree algorithm provides the best accuracy for Fmtstr with 0.01 difference. For all the classifiers, we selected the Random Forest algorithm because decision tree classifier can result in a complex tree that is incapable of generalizing the model which in turn results in overfitting. Random Forest Classifier is a tree-based classifier. It generates a collection of decision trees that fits the dataset on several sub-sets and utilizes averaging to avoid overfitting.

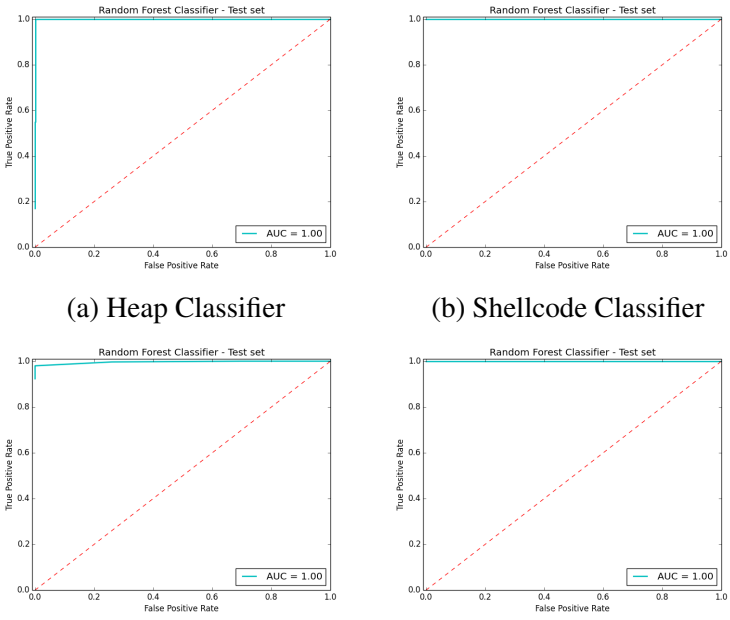
3.3 Classifiers Evaluation

Our tool does not only identify malicious behaviors after they are launched; it also able to identify the existence of any malicious behaviors. We validate our Classifiers on a validation set that is used as a test set while assessing the algorithms using cross-validation.

Table 3.8: Summary of the misclassified crash reports on the training, validation, and testing sets.

Set	Training	Validation	Test
Heap Classifier	1.46%	1.57%	1.63%
Shellcode Classifier	0%	0%	0%
Fmtstr Classifier	1.12%	0.86%	0.38%
ROP Classifier	0.23%	0.27%	0.05%

We further measure the accuracy of the classifiers on new, unseen data. Table 3.7 and 3.8 show the accuracy score and the misclassified rate respectively on training, validation, and testing sets for all the Classifiers. Figure 3.1 shows the (Receiver Operating Characteristics and the Area Under The Curve for each Classifier on the test sets.



(a) Heap Classifier

(b) Shellcode Classifier

(c) Fmtstr Classifier

(d) ROP Classifier

Figure 3.1: ROC curves for Random Forest Classifiers - Heap, Shellcode, Fmtstr and ROP Classifiers

3.3.1 Features Importance

To further assess our Classifiers, we measure the importance of the features for each classifier shown in Figure 3.2 to 3.5. To measure the significance of the features we used the weights of the features that are calculated by Gini impurity which is the function that is used to measure the quality of the Decision Tree split. Figure 3.6 shows the accuracy of ROP Classifier currently with no feature removed and when removing one feature at a time. We only removed the number of the code block, and the number of invalid chunks pointers features to highlight who removing them from the feature set decrease the accuracy.

3.4 System Evaluation

We evaluated our tool on some of the crash reports used during Classification phase and a new set of crash reports that have never been seen by the Classifiers. From the dataset used during Classification phase, we selected *ffmpeg* and *php* crash reports generated while

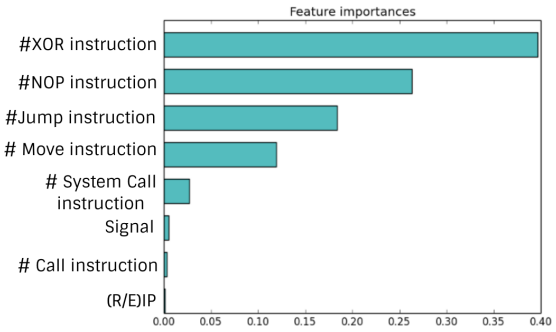


Figure 3.2: Featuers Importance - Shellcode Classifier

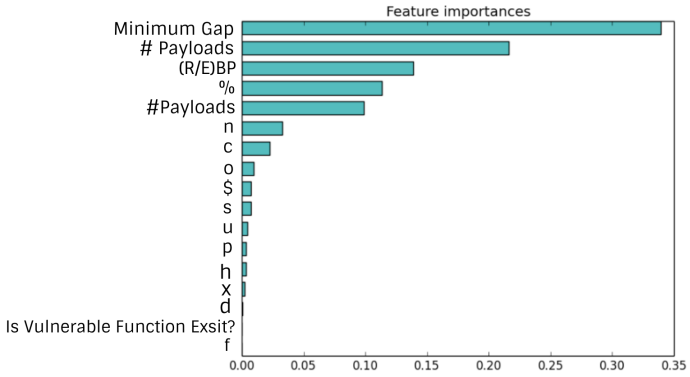


Figure 3.3: Featuers Importance - Fmtstr Classifier

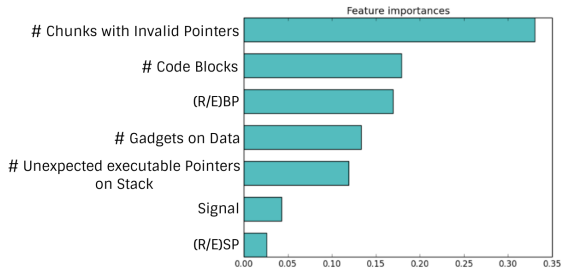


Figure 3.4: Featuers Importance - ROP Classifier

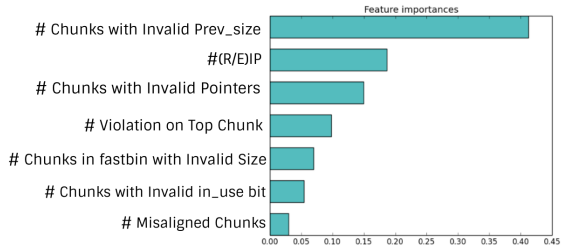


Figure 3.5: Featuers Importance - Heap Classifier

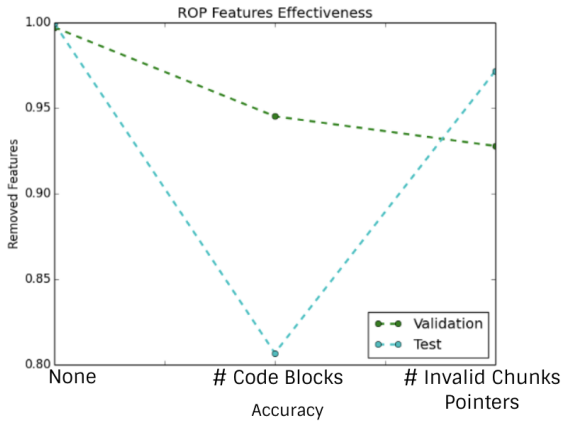


Figure 3.6: ROP Classifier - when removing two features: one feature at a time.

Table 3.9: Summary of the clustering performance on *ffmpeg* and *php* crash reports.

Binary	F-measure	Homogeneity	Silhouette Coefficient	Accuracy
ffmpeg	0.90	1	0.992	83%
php	1	1	0.997	100%

exploiting reported vulnerabilities in order to simulate real-world exploit. We evaluate the tool on the crash reports for *ffmpeg* targeting two vulnerabilities, CVE-2016-10190 and CVE-2016-10191. We further evaluate the tool on the crash reports for *php* targeting also two vulnerabilities, CVE-2015-8617 and CVE-2016-4071. In addition to evaluating the clustering method on *ffmpeg* and *php* crash reports and evaluating the classifiers on the validation and test set, we further evaluate the tool on unseen crash reports generated by new binary, *tachikoma* in order to ensure the effectiveness of our tool. *tachikoma* is one of the challenges on *DEF CON CTF Final 2015*.

ffmpeg crash reports: We run the completed tool on *ffmpeg* malicious crash reports in order to evaluate the correctness of our clustering technique. The Classifiers are trained on these crash reports and thus, have performed well with 100% accuracy and 0% misclassification. The tool clustered the crash reports into three clusters; one for crash reports caused by exploiting cve-2016-10191, and the other two are for crash reports caused by exploiting cve-2016-10190. The reason behind clustering cve-2016-10190 crash reports into two cluster is that the first cluster contains crash reports that crashed on trying to exploit

Table 3.10: Summary of the clustering results for *tachikoma* crash reports.

Cluster	Number of Crash Reports
1	3037
2	1487
3	544
4	55
5	6
6	1

the heap. However, the second cluster contains the already launched exploit. We evaluate the clustering results using the F-measure and Silhouette Coefficient and Homogeneity highlighted on Table 3.9

php crash reports: We yet evaluate the tool on *php* malicious crash reports to ensure the accuracy of the clustering results. The Classifiers are also trained on these crash reports and performed well with 100% accuracy and 0% misclassification — the crash reports clustered the crash reports into two clusters: one for cve-2016-4071 and the other for cve-2015-8617. We evaluate the clustering results using the F-measure and Silhouette Coefficient and Homogeneity shown in Table 3.9.

tachikoma crash reports: We further assess the tool on *tachikoma* crash reports containing 5130 crash reports. It is important to note that there were no crash reports from *tachikoma* on the dataset used on Classification phase. We recognize that all the crash reports caused by a binary programmed to be vulnerable and expected to be exploited. Therefore, considering the ground truth for all the crash reports of *tachikoma* to be malicious is a reliable assumption. Moreover, the functions that are vulnerable to Format String attack do not exist which give us another piece of ground truth to evaluate Fmtstr Classifier. Our tool classified all the crash reports as malicious; it classified 4986 as having Heap exploitation and shellcode injection and 144 as having only Shellcode injection. There no crash classified as having Format String attack which gives us 100% accuracy for Fmtstr Classifier. It then clusters all the crash reports, 5130, into six clusters summarized in Table 3.10.

3.5 System Performance

The performance of our tools relies on the performance of the feature extraction phase which in turn depends upon the size of the crashed binary. We measure the performance of the feature extraction phase when generating the dataset used on the Classification phase which has an average time of 7.4 seconds and a median of 1.06 seconds. We further measure the performance of our tool on *tachikoma* crash reports. The average time for extracting the features from *tachikoma* crash reports and classifying them is 0.022 millisecond, and the median is 0.021 millisecond and the time to cluster them is 5.93 seconds.

CHAPTER 4

CONCLUSION

In this work, we propose an effective method to identify whether crash reports caused by failed exploits and cluster them based on the exploited vulnerabilities using Machine Learning. Our evaluation results show that the tool is accurately classified and grouped attacker-driven crash reports.

REFERENCES

[1] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, IEEE, 2011, pp. 486–493.

[2] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, "Debugging in the (very) large: Ten years of implementation and experience," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ACM, 2009, pp. 103–116.

[3] M. Kerrisk. (2019). Linux programmer's manual, (visited on 09/30/2010).

[4] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: A method for clustering duplicate crash reports based on call stack similarity," in *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 1084–1093.

[5] R. el at, "Coroner: Attack-focused coredump analysis techniques," in *TBA*, Under Submission, TBA, TBA.

[6] D. Comaniciu and P. Meer, "Mean shift: A robust approach toward feature space analysis," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 5, pp. 603–619, 2002.

[7] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.