Exniffer: Learning to Rank Crashes by Assessing the Exploitability from Memory Dump

Thesis submitted in partial fulfillment
of the requirements for the degree of

MS in Computer Science & Engineering by Research

by

Shubham Tripathi
201407646
shubham.t@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
March 2018

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled **"Exniffer: Learning to Rank Crashes by Assessing the Exploitability from Memory Dump"** by **Shubham Tripathi**, has been carried out under my supervision and is not submitted elsewhere for a degree.

_____ _____

Date Adviser: Prof. Sanjay Rawat

“Contemplate and reflect upon knowledge, and you will become a benefactor to others.”

To my parents

Acknowledgments

I would like to express my gratitude to my adviser, Dr. Sanjay Rawat. Sanjay sir helped me staying focused on problems and provided new directions to approach them. Working with him, I have developed my problem solving skills, learnt about research, about life in general. I would be thankful to him for all my life, for providing me the guidance on various matters, always motivating and boosting me with confidence, which really helped me in shaping my life.

I must also thank all my lab-mates who are working or have worked earlier in CSTAR - Vijayendra, Spandan, Satwik, Charu, Teja, Ishan and Lokesh. I really enjoyed working with them in the lab. I would like to thank all my friends in IIIT, for making my stay in the campus a memorable one.

In the end, I would like to thank my parents, my wife Jagrati and my brother Utkarsh for their support and unconditional love and affection.

Abstract

An important component of software reliability is the assurance of certain security guarantees, such as absence of low-level bugs that may result in code exploitation, for example. A program crash is an early indicator of possible errors in the program like memory corruption, access violation or division by zero. In particular, a crash may indicate the presence of safety or security critical errors. A safety-error crash does not result in any exploitable condition, whereas a security-error crash allows an attacker to exploit a vulnerability. However, distinguishing one from the other is a non-trivial task. This exacerbates the problem in cases where we get hundreds of crashes and programmers have to make choices which crash to patch first!

In this work, we present a technique to identify security critical crashes by applying machine learning on a set of features derived from core-dump files and runtime information obtained from hardware assisted monitoring such as the last branch record (LBR) register. We implement the proposed technique in a prototype called *Exniffer*. Our empirical results, obtained by experimenting *Exniffer* on several crashes on real-world applications show that proposed technique is able to classify a given crash as *exploitable* or *not-exploitable* with high accuracy.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

During testing (or during the normal usage), a program crash indicates the presence of possible errors or bugs in the program. A program can crash because of many reasons like access violation, division by zero, unhandled exception etc. Generally, the application developers request for more information such as stack trace, memory dump etc. to diagnose and fix the problem. Figure 1.1 shows a typical scenario of such a crash. Kernel crashes are more serious and leads to unresponsive computer which has to be generally restarted. In windows this is known as the famous *Blue screen of death* as shown in Figure 1.2. Notice in the figure, the crash probably happened because of a page fault at certain memory addresses highlighted in the technical information. Also, after this exception the Windows OS started collecting data in the form of a crash dump for further analysis.

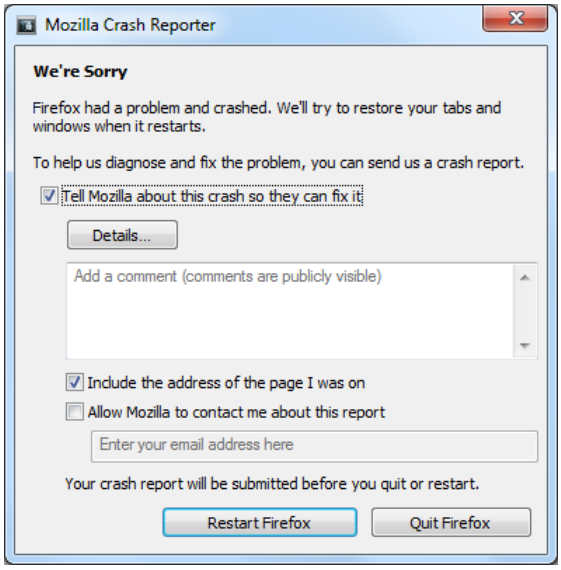

Figure 1.1: A typical crash report from Mozilla Firefox

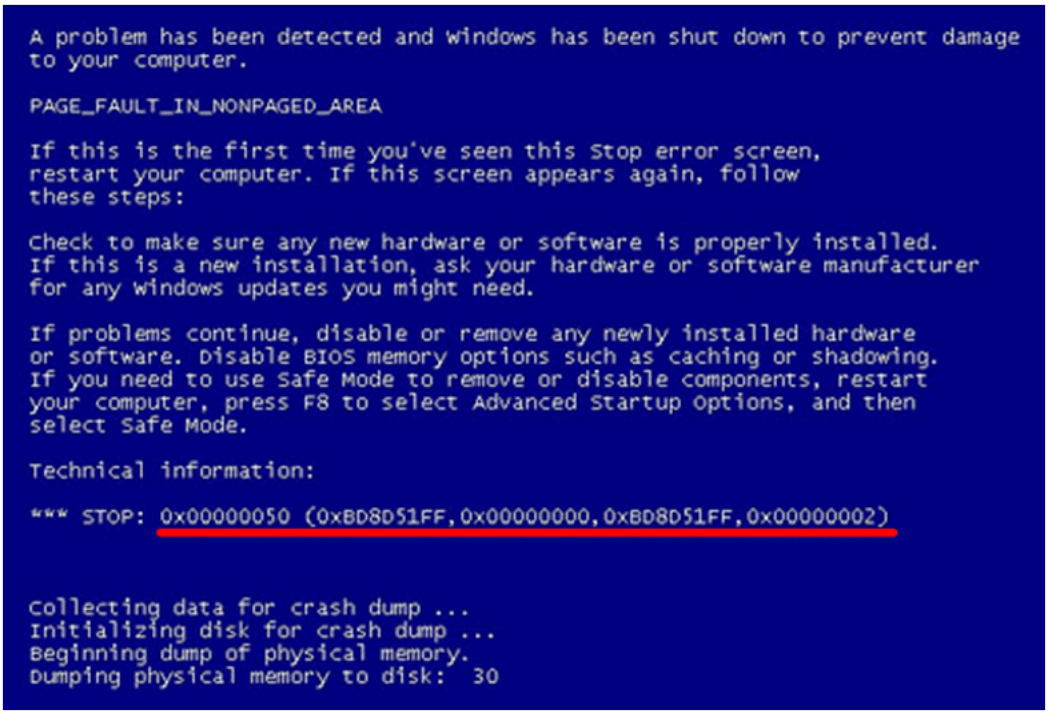

Figure 1.2: Windows OS crash: Blue screen of death

A program crash such as those shown in Figure 1.1 and 1.2 can be of two types, *safety* and *security*. A *safety* crash does not result in any exploitable condition, whereas a *security* crash allows an attacker to exploit a vulnerability such as buffer overflow. Listing 1.1 and 1.2 give examples of exploitable and non-exploitable crashes respectively.

Listing 1.1: Exploitable Example

```
1  int main(){
2          char name[30],ch;
3          int i=0;
4          printf("Enter␣name:␣");
5          while(ch!='\n') // terminates if user hit enter
6          {
7                  ch=getchar();
8                  printf("%d␣%c\n",i,ch);
9                  name[i]=ch;        // crash!
10                 i++;
11         }
12         printf("\n");
13         name[i]='\0';   // inserting null character at end
```

```
14          printf("Name: %s",name);
15          return 0;
16  }
```

Listing 1 shows a classic buffer overflow example that crashes at line 9. The code keeps accepting input characters in a buffer until the user hits enter. However, since the buffer has fixed size of 30 characters, a string with length larger than 30 will result in overwriting of the next set of addresses which is outside the memory allocated to buffer *name*. If the size of input string is large enough, it will lead to corruption of memory address containing the return address of the last function on the stack. Finally, when the current function returns, the corrupt value in return address leads to the segmentation fault in this program. Notice here that an attacker can provide a crafted input string that leads to a buffer overflow of the return address such that the control flow redirects back to memory containing buffer *name*. This crafted input stored in the buffer can contain instructions for shell code for example and therefore leads to its exploitation.

Listing 1.2: Non-Exploitable Example

```
1  void main() {
2          int a[4],i;
3          scanf("%d",&a);
4          a[i]=a[0]+a[1]+a[2]+a[3]; // crash!
5          printf("%d",a);
6  }
```

Listing 2 is an example of non-exploitable crash. Segmentation fault occurs at line 4 as a result of write access violation due to uninitialized variable *i* possibly containing some garbage value. Operation *a[i]* in line 4 tries to dereference the memory location *(a+i)* which is invalid, thus resulting in a segmentation fault. Notice here that there is no way an attacker can possibly control the value of variable *i*, which makes this crash non-exploitable.

From practical usage point of view, answering the question on exploitability has a very important role to play. If we look at any big product bug reporting site [3], we will notice that on an average, there are hundreds of bugs in the queue to be patched with the same limited resources to work on i.e. human developers. Under such conditions, if there is a way to prioritize the patch development, it can help the company to patch most important bugs fast. One of the ways to measure the importance of the patch is to find out if "it is possible to exploit the crash for further security breach". If we can assert the possibility of exploitation of a crash, it makes sense to immediately release a patch for it.

!exploitable [9] is one such tool that predicts exploitability of a crash by analysing crash state of a program within a debugger using specific rules and relates them using a rule-based engine. *!exploitable* however fails to generalize to different crash scenarios and conservatively assumes data available at

crash site as controlled by the attacker, which may not always be the case, and this leads to many false alarms. For example, *!exploitable* incorrectly predicts Listing 1.2 as exploitable.

1.2 Problem Statement

A core-dump provides the memory snapshot of the program at the time of crash and is used traditionally by the developers to determine root-cause and implications of a crash. However, information available from just core-dump may not be sufficient always for crash analysis and tools such as Crashfilter [12], ExpTracer [28] and Bitblaze [23] employ static and dynamic analysis to enrich the information available from a core-dump for a more accurate analysis. But such an analysis is really slow for complex programs and are not suited for real-time exploitability prediction such as in production environments. In addition, even though offline analysis can be performed by reproducing the crash, the conditions of crash may vary due to a change of environment. We would like to capture the crash information in the same environment in which the crash occurred.

Hardware debugging extensions available in recent commodity processors have been used to improve runtime for root-cause analysis in previous studies [17] [18]. For example, a particular extension of Intel i7 processor i.e. *Last Branch Record (LBR)* saves a set of few (generally 16) most recent branches and has been used to trace program flows and even determine code coverage. In this research work, we extract static features from core-dump and leverage LBR for dynamic features to enrich our feature set which makes our approach suitable for production systems.

Exploitability prediction of large number of crashes manually is a tedious task that requires effort, skill and resources which eventually costs money. Moreover, this process cannot be ignored as it can lead to serious ramifications for the organization and its customers. So, we propose Exniffer, an efficient and automated machine learning based tool for large scale exploitability prediction of crashes. Given sufficient samples (crashes) of the two classes i.e. exploitable and non-exploitable, Exniffer learns a hypothesis which can be used to classify an unseen sample. We use Support Vector Machine (SVM) for exploitability prediction. We utilize publically available dataset i.e. VDiscovery, LAVA and some online freely available for development, validation and evaluation of the hypothesis. We extract static features from core-dump and leverage LBR for dynamic features to enrich our feature set which makes Exniffer suitable for production systems. We only assume availability of a binary executable, a crashing input, a core-dump and the LBR records for exploitability prediction. Also, for this research work, we consider memory corruption vulnerabilities in *x86* architecture arising from C and C++ program executables only.

When we predict explotability for large number of crashes, there arises a need to assign priority to each crash that estimates how severe the crash is with respect to exploitability. This further supports the developer in fixing most relevant crashes first. So, we extend our exploitability prediction algorithm to

support crash prioritization also. We also report the most relevant features learnt by SVM hypothesis over the development dataset that contribute to exploitability prediction.

In this thesis, we predict exploitability of a crash in software application using machine learning through the use of core-dump, binary executable in C/C++, crashing input and hardware branch tracing facilities. We also prioritize crashes in decreasing order of exploitability to facilitate bug fixing.

1.3 Contribution

Specifically, we make following contributions:

1. Design of lightweight static (core-dump) and dynamic (LBR) features that enables online analysis of production-run application crashes. (First of its kind application of LBR to complement static features).
2. Exploitability prediction and crash prioritization using machine learning.
3. Feature ranking to provide insight into the causes of crash and thereby reasoning over exploitability.

1.4 Outline

The thesis is oulined as follows. In Chapter 4, we propose technique to solve the problem and describe terminology, feature design and exploitability prediction algorithm. We then evaluate, discuss and compare our prototype in Chapter 5. Next we list the related work done in Chapter 3. Finally we conclude in Chapter 6 and also provide future directions for this research work.

Chapter 2

Background

2.1 Program Crash

A program crash occurs when a computer program i.e. a software application or an operating system stops functioning properly and exits. The program generally become unresponsive after the crash and may even result in fatal system error (see Figure 1.1 and 1.2). If the application performs an operation that is not allowed by the operating system or hardware, the system raises an exception in the form of a signal or a fault in the application. A fault is a failure condition that can result in the program crash if the exception is not handled explicitly.

There are many types of exceptions such as segmentation fault, program abort, floating point error and illegal instruction etc. There are wide variety of causes of these exceptions. Memory corruption bugs are one of main reasons for program crash. In addition, some other causes include division by zero, operation on undefined values (for e.g. NaN), unhandled exceptions, malformed or unaligned instructions, invalid arguments to system calls, illegal I/O operation on hardware devices etc. Applications generally respond to the exceptions by dumping *core* files which consist of details of working memory at the time of crash (Section 2.2). These core-dump files can be used in conjunction with debuggers to diagnose the crash. From the point of view of exploitability, a program crash can be seen as a *denial of service* condition which may or may not be triggered by an attacker. If the attacker's input can reach the crash site, then attacker can influence the crash and may even exploit it. We use Dynamic Taint Analysis (Section 2.4) to determine exploitability of a crash by tracking the flow of input data through the program while it is executed. In the next section, we will discuss the different types memory corruption bugs that generally result in program crash.

2.1.1 Memory corruption

Memory corruption occurs if a program accesses memory in an unintended manner. This can result in a crash/exception if the memory access is illegal. However, many unintended memory operations can be legal which may not result in a crash. These kinds of corruption may change the state of the program so that it behaves unpredictably and possibly generates an exception at a later point in time. For example, reading an uninitialized variable is an unintended legal memory access because it may result in unpredictable behavior of the program depending on what the variable is used for. Dereferencing an uninitialized pointer, such as NULL pointer, will cause an exception immediately. But if the uninitialized pointer points to a unintended but valid memory location the program may not crash immediately and may lead of random behaviour at a later point in the program. Incorrect management of dynamically allocated memory can lead to another kind of memory corruption bug known as *use-after-free*. If a data structure or an object, pointed to by a pointer, is freed and that pointer is used again later, it can point to whatever has been allocated on the same address. This is known as a *use-after-free* bug. If the pointer is freed again, the program will try to free something that should not be freed, known as a *double-free* bug.

Memory corruption can also be caused by overwriting buffer memory which is known as *buffer overflow*. If there is insufficient boundary checking on the pointer values or indices when accessing buffers, memory outside the buffer could be accessed. If this memory is read, the situation is similar to reading uninitialized data. On the other hand, if data outside the buffer is written to, it may possibly overwrite other data. Since data buffers normally lie on the stack or the heap, a buffer overflow may overwrite important internal data structures as well as program variables. This can result in an exception at some point later depending on when the corrupted data is used. It can also result in immediate crash if an invalid memory or illegal memory is accessed while reading or writing buffer.

Type conversion can cause memory corruption if they are not accounted for in a program. For example a small negative integer will be interpreted as a huge positive number if it is treated as an unsigned integer. *Integer overflow* can cause the addition of two large numbers to result in a small number. These kinds of programming issues can lead to buffer overflows, for example by the incorrect calculation of a buffer length. *Concurrency* can also cause memory corruption. If memory is shared between different threads, there can be race conditions causing unintended use of memory. This can make one thread corrupt the variables of another thread. *Format string* bugs are caused by incorrect use of the C language format string functions like printf. A malicious user may use the format tokens such as %x and %n, to print data from the call stack or possibly other locations in memory.

2.2 Core-dump

A core-dump or crash-dump consists of the recorded state of the working memory of a computer program at a specific time, generally when the program has crashed or otherwise terminated abnormally. The core-dump also contains information about processor registers, such as program counter and stack pointer. It also contains memory management information such as information on memory segment size, addresses and permissions. A crash can be dumped either as a kernel dump that contains memory state of kernel or minidump that has only user level information or a full dump that contains dump of complete memory. Core-dump format is generally Executable and Linkable Format (ELF) for Linux and Portable Executable (PE) in case of Windows. In this project, we work with x86 ELF minidumps.

Figure 2.1 shows the two views of same x86 ELF executable: *Executable view* and *Linkable view*. Operating system uses segment information available from the executable view (program headers) to load the binary into memory and execute it. Linkable view (section header) on the other hand organize the binary into logical areas (sections) to communicate information between the compiler and linker. From a unified perspective, each segment consists of at least one section in an ELF file as is also evident from Figure 2.1.

Figure 2.2 shows header information in a typical ELF core file which consists of details such as ELF magic bytes, architecture, type of ELF file, entry point of binary, start of program and section header etc. If an ELF file is compiled with debugging option, it consists of a *symbol table* that references a *string table* consisting of variable and function names suitable for debugging purpose. A stripped ELF is not compiled with debugging option and does not contain any such information. Our approach does not assume or utilize any such debugging information to predict exploitability.

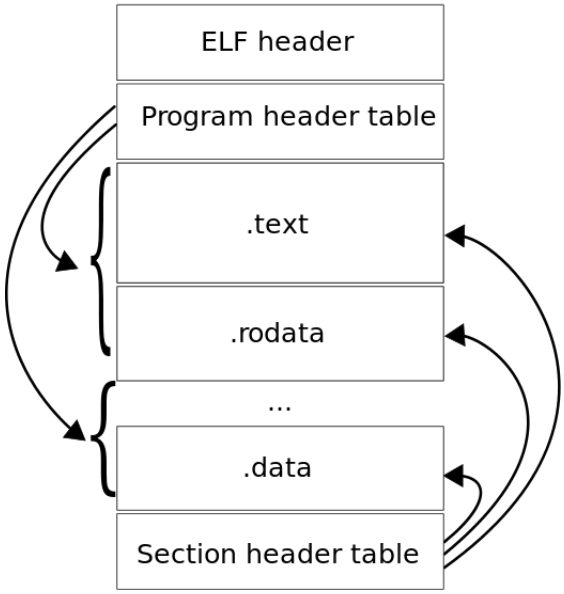

Figure 2.1: An ELF file has two views: the program header shows the *segments* used at run time, whereas the *section* header lists the set of sections of the binary.

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              CORE (Core file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          2328328 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         22
  Size of section headers:           40 (bytes)
  Number of section headers:         24
  Section header string table index: 23
```

Figure 2.2: ELF header: ELF file information

2.3 Last Branch Record (LBR)

Modern intel processors (post - Pentium 4) provide hardware support for debugging facilities for branch tracing as part of performance monitoring units in the form of Last Branch Record (LBR) and Branch Trace Store (BTS). LBR stores last few branches in the form of source and destination addresses using a circular ring of hardware registers. The number of records that LBR can store vary with the model of processor and are theore called Model Specific Registers (MSR) - it goes from 4 entries in Pentium 4 and Intel Xeon processors, to 8 in Pentium M processors, and to 16 in Nehalem processors [4]. LBR can also be configured to record different types of branch instructions, for example conditional branches, unconditional jumps, calls, returns etc. When enabled, LBR keeps a record of last N branch instructions and incurs almost zero overhead [17]. BTS on the other hand logs all the branch records in memory and as a result incurs more overhead than LBR. For our purpose, we utilize LBR to trace 16 branch instructions consisting of source and destination addresses. Figure 2.3 shows last 16 source and destination addresses stored in LBR along with the branch instruction.

```
0xb6238ec9 0xb6238158 ret
0xb6238188 0xb62384d0 call dword ptr [ecx+0x38]
0xb62384f1 0xb62a2130 jmp 0xb62a2130
0xb62a214c 0xb7fdd414 call dword ptr gs:[0x10]
0xb7fdd427 0xb62a2153 ret
0xb62a215b 0xb623818b ret
0xb623818e 0xb62381d8 jle 0xb62381d8
0xb62381e2 0xb62381b7 jmp 0xb62381b7
0xb62381ca 0xb62393a9 ret
0xb62393ac 0xb62393c0 jz 0xb62393c0
0xb62393c5 0xb62393ba jmp 0xb62393ba
0xb62393be 0xb62391c0 ret
0xb62391c5 0xb622f7f4 ret
0xb622f7f6 0xb622f7a8 jmp 0xb622f7a8
0xb622f7ce 0xb622f7d1 jz 0xb622f7d1
0xb622f7eb 0x80484f6 ret
```

Figure 2.3: LBR: Last Branch Records example

2.4 Dynamic Taint Analysis

Exploitability prediction using machine learning requires a dataset for training, validation and testing. Each sample in the dataset has to be labelled with one of the classes considered for machine learning hypothesis. For example, in our case, we consider exploitability prediction as a binary classification problem, so we have *two* classes i.e. Exploitable and Non-Exploitable. Now, to label a crash in one of the two classes, we need mechanisms to determine its exploitability. We utilize Dynamic Taint Analysis (DTA) as one such mechanism to determine exploitability of a crash for the purpose of dataset

labelling.

DTA is a form of data flow analysis to track data as it flows through a program while it is executed. DTA works at binary level and can perform taint analysis for dynamic libraries with exact runtime information such as register and memory values. This is in contrast to Static Taint Analysis (STA) which performs execution symbolically and precise runtime information is unavailable. This makes STA more suited to full coverage analysis. The tracked tainted information in DTA can be bit or byte level. Bit-level taint analysis is more precise but slower than byte-level analysis. The steps for DTA are as follows:

1. *Taint Initialization:* The input to the program given by user or an untrustworthy source is marked as tainted. Actually, the memory addresses containing the tainted input is essential to perform DTA and stored in a buffer or a file. Generally, system calls such as read are hooked and memory locations where the input data is stored is captured and tracked.

2. *Taint Propagation:* Propagation of taint is performed in three steps. First, a memory address containing the tainted data is read using a load instruction such as `mov [%eax],%ebx`. The register in which the tainted data is read is now marked as tainted i.e. in the above example, `%ebx` is marked as tainted. So, there are two sets of taint information stored: memory addresses of tainted data and registers into which this tainted data is loaded. Second, memory write instructions such as mov `%eax,[%esp]`, results in tainting of memory address if the register which is to be written, is tainted. In the above example, if `%eax` is tainted, then memory address of the operand `[%esp]` will now contain tainted data. Third, boolean and arithmetic instructions such as `mul %eax, %ecx` also result in propagation of taint information. In this example, if `%ecx` is tainted and `%eax` is not tainted, then after the operation `%eax` will be marked as tainted.

Figure 2.4 describes an example of the flow of tainted information. Data from tainted memory location is moved to register `%xmm2` with a memory read instruction. `%xmm2` is marked as tainted. The taint then propagates to `%eax` when the data in `%xmm2` is moved to `%eax`. With a memory write, the tainted data in `%eax` is now written to a memory location. This memory location is now stored in a buffer to keep track of the tainted data it holds. The process of taint propagation now repeats when a memory read instruction moves the tainted data to `%esi`.

In the context of program crash analysis, DTA can be used to decide if the corrupted memory generating a crash originates from user input or not. For example, if a program crashed with segmentation fault at a memory read instruction, and if the memory address read contains tainted data, then the attacker will be able to control the register value and may possibly exploit the bug. This is a basic factor to decide the exploitability of a crash. DTA although promising, involves runtime and memory overhead and its usage in production systems for exploitability analysis can be a costly affair. Therefore, in this work, we utilize DTA only for the labelling of the dataset and not for crash analysis.

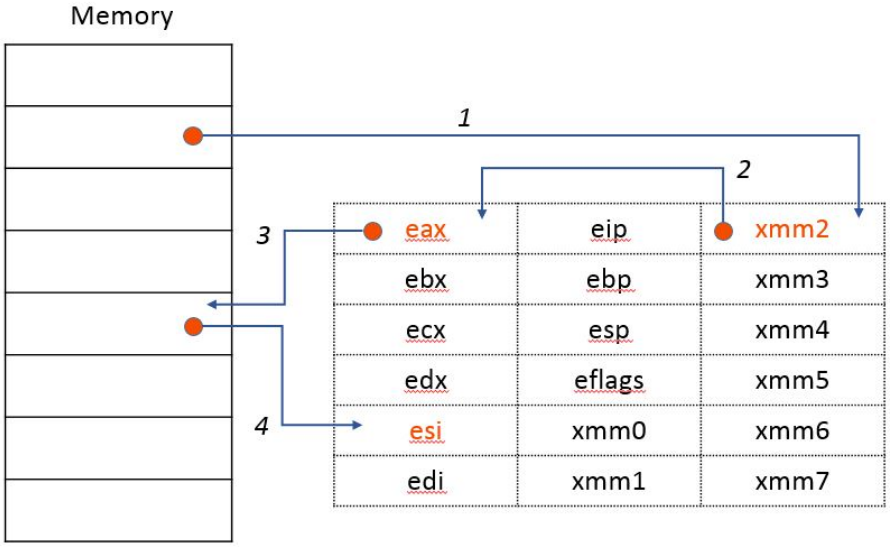

Figure 2.4: Dynamic Taintflow Analysis: Taint Propagation

2.5 Support Vector Machine

We pose the exploitability prediction as a binary classification problem with two classes. The features for each crash are extracted from core-dump and LBR. A simplified approach involves building a linear hyperplane that separates the two classes. But it is rare for a real-world dataset to be linearly separable. If however, the dataset is mapped to an appropriate higher dimensional space, it can become linearly separable in that feature space. Although a promising insight, there are two major issues. First, there is an explosion in the number of dimensions of higher dimension feature space. For e.g. a p^{th} degree polynomial discriminant function in the original space $\mathbb{R}^m$ requires a feature space of $O(m^p)$. This results in huge computational cost for both learning and final operation of classifier. And secondly, since we are learning $O(m^p)$ rather than $O(m)$ parameters, we may need much larger number of examples for achieving proper generalization.

Fortunately, Support Vector Machine (SVM) offers an elegant solution to both the issues. SVM learns an optimal separating hyperplane that maximizes separation between two classes. It can also learn non-linear discriminant functions by effectively mapping original input space to high-dimensional feature space. Furthermore, by using *kernel functions*, we never explicitly compute the mapping of training or testing feature vectors.

Following section formally describe SVM hypothesis.

Let $X_i \in \mathbb{R}^m$ represent a sample in n training examples, $y_i \in \{-1, +1\}$ be the label of each sample, $\phi : \mathbb{R}^m \rightarrow \Re$ be the mapping from input vector space $\mathbb{R}^m$ to feature space $\Re$ and $\langle W \in \Re, b \in \mathbb{R} \rangle$ be

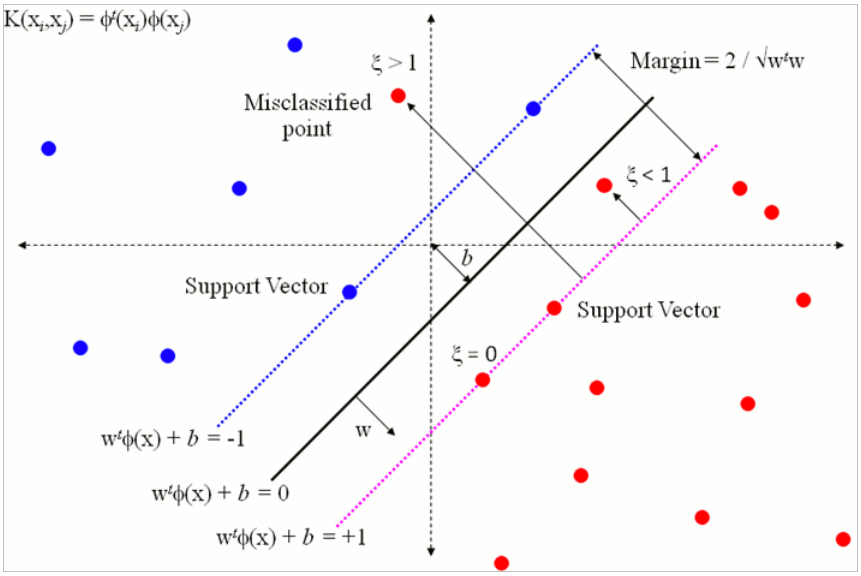

Figure 2.5: Soft margin Support Vector Machine with Kernel function

the maximum margin optimal hyperplane, then the separability constraints for the an SVM are given by,

$$y_i(W^T\phi(X_i) + b) \geq 1 - \xi_i \tag{2.1}$$

$$\xi_i \geq 0 \tag{2.2}$$

where $\xi_i \in \mathbb{R}$ are the slack variables corresponding to each training sample and measure the error with respect to the optimal hyperplane. This results in a soft-margin SVM (Figure 2.5) with optimization objective as minimization of,

$$\frac{1}{2}W^TW + C\sum_{i=1}^{n}\xi_i \tag{2.3}$$

subject to constraints given in (2.1) and (2.2) where i=1...n and C is the user defined regularization parameter. It is possible to prove [Vapnik 1998] that optimum hyperplane (W^*, b^*) is given by,

$$W^* = \sum_{i=1}^{n}\mu_i^* y_i \phi(X_i) \tag{2.4}$$

$$b^* = y_j - W^{*T}\phi(X_j)\text{ , j such that } 0 < \mu_j < C \tag{2.5}$$

here μ_i's are positive real numbers that maximize the $dual$ of the optimization objective,

$$\sum_{i=1}^{n}\mu_i - \frac{1}{2}\sum_{i,j=1}^{n}\mu_i\mu_j y_i y_j \phi(X_i)^T\phi(X_j) \tag{2.6}$$

subject to,

$$\sum_{i=1}^{n} \mu_i y_i = 0 \text{ and } 0 \leq \mu_i \leq C \tag{2.7}$$

The class of a test sample X can now be predicted as,

$$sign(W^{*T}\phi(X) + b^*) \tag{2.8}$$

$$sign(\sum_{i=1}^{n} \mu_i^* y_i \phi(X_i)^T \phi(X) + b^*) \tag{2.9}$$

From Eq.2.9, notice that there is only a subset of training samples X_i's with $\mu_i^* > 0$ that are used for prediction. These samples are known as *support vectors*.

ϕ maps X to some higher dimensional space $\Re$ which has to be computed. However, it turns out that if there is a function $K(X_i, X_j) = \phi(X_i)^T \phi(X_j)$, it is possible for SVM to learn and also use maximum margin hyperplane efficiently in $\Re$ without computing the mapping explicitly [Mercer Theorem]. $K(X_i, X_j)$ is known as the *kernel function*. For our purpose, we have created models using SVM with a linear kernel and with a *radial basis function (rbf)* kernel. A linear kernel is a special kernel where $K(X_i, X_j) = X_i^T X_j$ while an rbf kernel is gaussian function $K(X_i, X_j) = \exp(-\gamma |X_i - X_j|^2)$ where $\gamma > 0$ is a user-specified parameter that is inversely proportional to the variance of the gaussian function.

2.5.1 Effect of parameters C and γ on model selection

The regularization parameter C is the weight of sum of errors made by the maximum margin hyperplane $\langle W, b \rangle$. If C is too high, then for expression 2.3 to be minimized, the term $\sum_{i=1}^{n} \xi_i$ has to very small, which means that the selected model makes little or no errors in classfication and hence overfits the data. On the other hand, if C is very small, then ξ_i's can take any value and W underfits the data making a lot of errors.

γ, which is a parameter for *rbf* kernel, controls the area of influence of samples selected by model as support vectors. A high γ signifies a low variance i.e. the area of influence of support vector only includes support vector itself which results in overfitting. When γ is too small, then the influence of support vector is large and the model will not be able to capture the complexity of the data.
Therefore, C and γ need to be set in such a way so as to prevent both under and over-fitting.

Chapter 3

Related Work

!exploitable and its adaptations for other operating systems [13] performs exploitability analysis of crashes. The tool first creates hashes to determine the uniqueness of a crash and then assigns an exploitability rating to the crash: Exploitable, Probably Exploitable, Probably Not Exploitable, or Unknown. The tool reads the state of the crash within Windbg after the crash occurs and uses a rule based engine to determine the exploitability rating. Since exploitable crashes can manifest in a variety of ways, capturing all the rules is not feasible task and the rule-set has to be constantly evolved. In addition, the tool conservatively assumes all the input available at the point of crash point as tainted which results in reduced accuracy.

Crashfilter [12], another tool that performs program tracing from crash point to exploitable point (branch instruction) by using static taint analysis for exhaustive exploitable point search. However, similar to !exploitable, it assumes that inputs available at crash point are all tainted and analyze ARM executables only. In addition, static taint analysis also has large runtime overhead.

VDiscover [14] shares the same goal as Exniffer, in extracting lightweight static and dynamic features and predicting exploitability using a machine learning model. Dynamic features are extracted as sequence of calls to C standard library. Multiple such sequences are extracted by incorporating fuzzing to generate malformed inputs. We found that to correctly predict on a testcase, VDiscover requires long sequences of dynamic features which may not always be the case. VDiscover has a low false positive rate, however there is scope of improvement with respect to true positive rate.

ExploitMeter [21] integrates fuzzing for testcase generation with machine learning for quantifying software exploitability. However, they use !exploitable to label their dataset and as we have seen in comparison section 5.2.5, !exploitable results in a lower accuracy. This implies that if Exniffer performs better than !exploitable, it will perform better than ExploitMeter on a given dataset. ExploitMeter uses mostly static features from binary executables extracted using hexdump, objdump and readelf utilities. The authors clearly observed weak predictive power of these features and infact VDiscover also indi-

cated ineffectiveness of similar static features.

Kim et. al. [20] use machine learning for the purpose of crashes prioritization. The authors focus on determining top few crashes that account for large majority of crash reports, although they do not consider exploitability of crashes in their work.

CREDAL [29] aims to localize memory corruption vulnerabilities using information from core dump such as stack traces and combines it with source code to give more insight to developer by highlighting code fragments that correspond to data corruption. We do not use source code for our analysis and our idea is to provide a fast screening of large number of crashes to prioritize debugging efforts.

Several systems are developed that perform automated root cause analysis and exploit generation such as MAYHEM [19] and Lai et al. [16] performs automatic exploit generation with concolic execution while Heelan et al. [15] use taint-flow analysis given crashing input and binary executable. Miller et al. [23] analyse crashes with whole system taint-tracking and manual analysis to determine accurately if the crash is indeed exploitable. It is true that if one is able to generate an exploit for a crash, it is definitely exploitable. However, since the runtime to generate execution traces (in taint-flow analysis for example), to generate exploit is very large, we propose that these approaches be used for complete analysis of only most severe crashes such as those predicted exploitable by Exniffer.

In their paper on predicting top crashes, Park et al. [20] hypothesized that only a small number of top crashes account for majority of crash reports. They extracted features from relevant methods and stack traces related to history, complexity metrics and social network analysis to train a machine learning model for predicting top crashes. Although their work provides a prioritization of crashes, they do not look into exploitability prediction.

Tang et al. [30] use reverse taint analysis from instruction pointer to user influenced inputs using shadow memory to identify the root cause of the vulnerability leading to the crash. They also generate exploit for confirming exploitability of a crash using symbolic execution. As we discussed, such an approach can help perform a detailed exploitability analysis of few chosen crashes. RETracer [10] also performs binary-level backward taint analysis, but instead of using shadow memory, they reconstruct program semantics using core dump. However, RETracer does not deal with exploitability of crash at this point.

Directed greybox fuzzing [25] uses a simulated annealing based approach to patch testing, crash reproduction, static analysis report verification and information flow detection. This approach is suitable for offline analysis and requires source code to perform compile-time instrumentation.

DeepLog [26] performs anomaly detection and diagnosis from system logs using deep neural network to learn log patterns from normal execution and detect anomalies when log patterns deviate. DeepLog can detect crashes real-time in production, but do not analyze or predict exploitability. For e.g. for either exploitable or non-exploitable crash, DeepLog will report a segmentation fault.

Failure Sketching [18] is a technique for Automated Root Cause Diagnosis of In-Production Failures. The system relies on hardware watchpoints and a new hardware feature for extracting control-flow traces using Intel Processor Trace (Intel PT) with low overhead of 3.74%. Failure Sketching uses source code since the objective is to help developer determine root cause.

Leveraging the Short-Term Memory of Hardware to Diagnose Production-Run Software Failures [17] transforms source code to toggle LBR at specific points in program to enable profiling of execution at runtime. Developers can use the LBR record collected at a failure site to reconstruct the control flow and interleaving right before the failure. This approach argues that short term memory of program execution is often sufficient for failure diagnosis.

Chapter 4

Proposed Technique

4.1 Feature Design

Machine learning requires crashes to be represented in features vectors. These features have to be designed and then extracted from core-dump and LBR for each crash. The design of features must ensure efficient extraction, address space invariance and relevance to memory corruption vulnerabilities. In particular, we extract two types of features, static and dynamic from core-dump and LBR respectively. Table 4.1 lists all the features that describe a feature vector.

4.1.1 Static Features

1. *Stack Unwinding*: Unwinding of stack, traces back functions (activation records) that were active at the time of crash. This is popularly known as *backtrace* in debugging terminology. In x86 architecture, the base pointer register is used to trace back currently active activation records known as frames. The base pointer register points to the location in memory that contains the address of base pointer of the last frame or the caller of current frame. If we use this to trace back each frame's base pointer, we would eventually reach the first frame which will have the address `0x00000000`. We then stop unwinding the frames any further.
   Buffer overflow in the current frame may lead to corruption of the memory location pointed to by the base pointer, which results in incorrect or unsuccessful unwinding of frames. So, we will use a boolean feature representing corruption of stack.

2. *Special Registers*: The x86 architecture provides special registers in the form of *instruction pointer*, *base pointer* and *stack pointer* that keep track of the next instruction to be executed, address of the last frame and top of stack respectively. We are interested to know if these registers point to valid memory locations or not. Valid memory locations are those segments that are allocated for the process by the operating system and some of these segments which include stack and heap may keep changing as the program executes. If any memory is accessed outside of these allocated segments, it will result in a crash known as *segmentation fault*. In addition to the legal

segments which represent the allocated memory, the validity of a segment is also defined by its permission flags i.e. *read, write and execute*.
So, an instruction pointer register must always point to the code segment which has the read and execute permissions and never to a segment with read, write and execute permissions. Also, segments containing stack must have only read and write permissions and must not have execute permission. An executable stack has been a common cause of shell code injection attacks. So, we design features to capture information related to special registers and memory segmentation.
Figure 4.1 shows the typical section information available in a *core-dump* file extracted using GDB [2]. Note that the sections are synonymous with segments in this case, however, in general there can be multiple sections in a segment. The first and second columns provide the valid address range for each segment at the time of crash. For example, `[0xbffdf000, 0xc0000000)` is the address range of the last segment which is generally part of stack. In the Listing 4.1, the program crashed at instruction:

Listing 4.1: Crashing instruction at Invalid address

```
(gdb) x/i $eip
=> 0x8048505 <main+57>: mov    %al,(%edx)
(gdb) info registers $edx
edx            0xc0000000       -1073741824
```

Notice that at the crashing instruction, the contents of register `al` were being moved to the memory address pointed to by `edx` register which was `0xc0000000`. But since, this is an out of bounds of the last section as shown in figure 4.1, the program crashes as a result of segmentation fault.
We also check the permissions of segment pointed to by *eip* register, which can be extracted from *Readelf* command line utility as shown in figure 4.2. *eip* points to `0x8048505` that lies in the segment starting at virtual address `0x08048000`. Notice that this segment has *Read* and *Execute* permissions as indicated by *R* and *E* respectively in the *Flg* column which is generally associated with *code* section. So, using this information we formulate binary features that capture validity of address and segment permissions of addresses contained in special registers and memory operand (see list of features in table 4.1).

3. *Crashing Instruction*: The instruction at which the program has crashed contains important information regarding the type of access violation. A memory read instruction represents read access violation while a memory write instruction represents write access violation. We include the type of access violation in the feature set. We also check whether the operands are contained within memory segments allocated at the time of crash. In addition, whether the type of instruction is branch or not is important to capture if there was a control flow change at the time of crash.

```
Core file:
    `/home/l0n3r/crash/downloaded/C/cGZDIT/cGZDIT.core', file type elf32-i386.
    0x0000->0x03fc at 0x000002f4: note0 READONLY HAS_CONTENTS
    0x0000->0x0044 at 0x000003e0: .reg/11127 HAS_CONTENTS
    0x0000->0x0044 at 0x000003e0: .reg HAS_CONTENTS
    0x0000->0x0200 at 0x0000043c: .reg-xfp/11127 HAS_CONTENTS
    0x0000->0x0200 at 0x0000043c: .reg-xfp HAS_CONTENTS
    0x0000->0x00a0 at 0x00000650: .auxv HAS_CONTENTS
    0x8048000->0x8049000 at 0x000006f0: load1 ALLOC LOAD READONLY CODE HAS_CONTENTS
    0x8049000->0x804a000 at 0x000016f0: load2 ALLOC LOAD READONLY HAS_CONTENTS
    0x804a000->0x804b000 at 0x000026f0: load3 ALLOC LOAD HAS_CONTENTS
    0xb7dd2000->0xb7dd3000 at 0x000036f0: load4 ALLOC LOAD HAS_CONTENTS
    0xb7dd3000->0xb7f72000 at 0x000046f0: load5 ALLOC LOAD READONLY CODE HAS_CONTENTS
    0xb7f72000->0xb7f74000 at 0x001a36f0: load6 ALLOC LOAD READONLY HAS_CONTENTS
    0xb7f74000->0xb7f75000 at 0x001a56f0: load7 ALLOC LOAD HAS_CONTENTS
    0xb7f75000->0xb7f79000 at 0x001a66f0: load8 ALLOC LOAD HAS_CONTENTS
    0xb7f79000->0xb7f90000 at 0x001aa6f0: load9 ALLOC LOAD READONLY CODE HAS_CONTENTS
    0xb7f90000->0xb7f91000 at 0x001c16f0: load10 ALLOC LOAD READONLY HAS_CONTENTS
    0xb7f91000->0xb7f92000 at 0x001c26f0: load11 ALLOC LOAD HAS_CONTENTS
    0xb7f92000->0xb7f94000 at 0x001c36f0: load12 ALLOC LOAD HAS_CONTENTS
    0xb7f94000->0xb7fbe000 at 0x001c56f0: load13 ALLOC LOAD READONLY CODE HAS_CONTENTS
    0xb7fbe000->0xb7fbf000 at 0x001ef6f0: load14 ALLOC LOAD READONLY HAS_CONTENTS
    0xb7fbf000->0xb7fc0000 at 0x001f06f0: load15 ALLOC LOAD HAS_CONTENTS
    0xb7fda000->0xb7fdd000 at 0x001f16f0: load16 ALLOC LOAD HAS_CONTENTS
    0xb7fdd000->0xb7fde000 at 0x001f46f0: load17 ALLOC LOAD READONLY CODE HAS_CONTENTS
    0xb7fde000->0xb7ffe000 at 0x001f56f0: load18 ALLOC LOAD READONLY CODE HAS_CONTENTS
    0xb7ffe000->0xb7fff000 at 0x002156f0: load19 ALLOC LOAD READONLY HAS_CONTENTS
    0xb7fff000->0xb8000000 at 0x002166f0: load20 ALLOC LOAD HAS_CONTENTS
    0xbffdf000->0xc0000000 at 0x002176f0: load21 ALLOC LOAD HAS_CONTENTS
```

Figure 4.1: Valid memory addresses: Core file contents displayed using GDB shows the address range of loaded sections at the time of crash

We also include the number of operands and type of each operand (memory, register, immediate) in the feature list so that the learning algorithm can distinguish the type of instructions. For example, a program crashed due to division by zero at `div ecx`. Suppose, another program crashed at `mov eax,[ecx]`. In both cases, `ecx` register contains zero but the crash in the first case is because of division by zero and in the second case because of illegal memory access. This scenario can be differentiated using a feature that checks if the crashing instruction contains an operand that accesses memory.

4. *Type of Signal*: We extract features from core-dump that include information on crashing signal such as benign, malformed instruction, access violation, abort and floating point exception. Malformed instruction signal is sent to the process when it attempts to execute an illegal, malformed, unknown, or privileged instruction. The segmentation fault signal is sent to a process when it makes an invalid virtual memory reference. The abort signal is usually initiated by the process itself when it calls abort() function in the C library for example due to incorrect memory allocation. The floating point exception signal is sent to a process when it executes an erroneous arithmetic operation, such as division by zero. The rest of the signals are considered as benign.

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  NOTE           0x0002f4 0x00000000 0x00000000 0x003fc 0x00000 R   0x1
  LOAD           0x0006f0 0x08048000 0x00000000 0x01000 0x01000 R E 0x1
  LOAD           0x0016f0 0x08049000 0x00000000 0x01000 0x01000 R   0x1
  LOAD           0x0026f0 0x0804a000 0x00000000 0x01000 0x01000 RW  0x1
  LOAD           0x0036f0 0xb7dd2000 0x00000000 0x01000 0x01000 RW  0x1
  LOAD           0x0046f0 0xb7dd3000 0x00000000 0x19f000 0x19f000 R E 0x1
  LOAD           0x1a36f0 0xb7f72000 0x00000000 0x02000 0x02000 R   0x1
  LOAD           0x1a56f0 0xb7f74000 0x00000000 0x01000 0x01000 RW  0x1
  LOAD           0x1a66f0 0xb7f75000 0x00000000 0x04000 0x04000 RW  0x1
  LOAD           0x1aa6f0 0xb7f79000 0x00000000 0x17000 0x17000 R E 0x1
  LOAD           0x1c16f0 0xb7f90000 0x00000000 0x01000 0x01000 R   0x1
  LOAD           0x1c26f0 0xb7f91000 0x00000000 0x01000 0x01000 RW  0x1
  LOAD           0x1c36f0 0xb7f92000 0x00000000 0x02000 0x02000 RW  0x1
  LOAD           0x1c56f0 0xb7f94000 0x00000000 0x2a000 0x2a000 R E 0x1
  LOAD           0x1ef6f0 0xb7fbe000 0x00000000 0x01000 0x01000 R   0x1
  LOAD           0x1f06f0 0xb7fbf000 0x00000000 0x01000 0x01000 RW  0x1
  LOAD           0x1f16f0 0xb7fda000 0x00000000 0x03000 0x03000 RW  0x1
  LOAD           0x1f46f0 0xb7fdd000 0x00000000 0x01000 0x01000 R E 0x1
  LOAD           0x1f56f0 0xb7fde000 0x00000000 0x20000 0x20000 R E 0x1
  LOAD           0x2156f0 0xb7ffe000 0x00000000 0x01000 0x01000 R   0x1
  LOAD           0x2166f0 0xb7fff000 0x00000000 0x01000 0x01000 RW  0x1
  LOAD           0x2176f0 0xbffdf000 0x00000000 0x21000 0x21000 RW  0x1
```

Figure 4.2: Segment Permissions: Readelf Linux utility shows the permissions available on each segment (See column *Flg*, R:Read, W:Write, E:Execute)

4.1.2 Dynamic Features

Hardware features such as LBR (Last Branch Record) provided by modern commodity processors incur negligible overhead and are used in root cause and security analysis. LBRA/LCRA [17] have also argued that for failure diagnosis, short-term program execution memory is sufficient. Using this information, we extract dynamic features based on LBR which are not only efficient but are also address invariant. Specifically, we extract the following features:

1. *Type of Branch instruction executed last*: Static features already capture if the program crashed at a branch instruction, which can lead to control flow redirection. This feature provides a context before the crash if the crashing instruction is not a branch instruction. The branch instruction considered are conditional or unconditional jump, function call and return.

2. *Occurence of crash in a loop*: Vulnerabilities such as stack and heap overflow have a high chance of occurring inside a loop where memory is sequencially written [24]. We find patterns within the branch records to look for a repeating address within the context of the last function. If the instruction is conditional or unconditional jump and destination address is smaller than the source address, then there is a possible repetition of instruction and can be considered as a loop.

3. *Number of call-return pairs*: This feature represents how frequently the functions are called and returned within 16 branch records. This feature captures the recursive context before the crash.

4. *Branch variation*: This feature is computed by variance of difference of source and destination address of each branch instruction available in LBR. This represents the variation of branch instructions. For example, a dynamic library which is loaded between stack and heap region has a large virtual address. When the user code in the text section with relatively small virtual address calls such a library function, the difference between the source and destination addresses is very large which contributes to a large variance. This feature captures the spread of control flow over the virtual memory space of the program. Since this is a positive real number feature with a large value, we take the absolute value of the logarithm of the variance.

Table 4.1: List of all features

Index	Description
	Static Features based on Core dump
	Stack Trace
1.	Backtrace is Corrupt
	Special Registers
2.	EIP is in Allocated memory
3.	EBP is in Allocated memory
4.	ESP is in Allocated memory
	Instruction and Operand
5.	Current instruction is available
6.	EIP Segment is Readable
7.	EIP Segment is Writable
8.	EIP Segment is Executable
9.	EIP Segment is Write $\oplus$ Execute
10.	Memory operand is in Allocated memory
11.	Memory operand is Source
12.	Memory operand is Dest
13.	Memory operand is Null
14.	#Operands = 0
15.	#Operands = 1
16.	#Operands = 2
17.	#Operands = 3+
18.	Operand is memory
19.	Operand is immediate

20. Operand is register
21. Operand is real number
22. Is Branch Instruction

Eflags Register

23. Carry Flag
24. Parity flag
25. Auxiliary Carry Flag
26. Zero Flag
27. Sign Flag
28. Trap Flag
29. Interrupt Enable Flag
30. Direction Flag
31. Overflow Flag
32. Input/Output privilege level flags
33. Nested Task Flag
34. Resume Flag
35. Virtual 8086 Mode flag
36. Alignment check flag (486+)
37. Virtual interrupt flag
38. Virtual interrupt pending flag
39. ID flag

Signals

40. Benign Signal
41. Malformed Inst Signal
42. Access Voilation Signal
43. Floating point exception Signal
44. Abort Signal

Dynamic Features based on LBR

45. Unconditional Jump
46. Conditional Jump
47. Return Instruction
48. Function Call
49. Loop
50. #Call-Return Pairs
51. Branch Variation

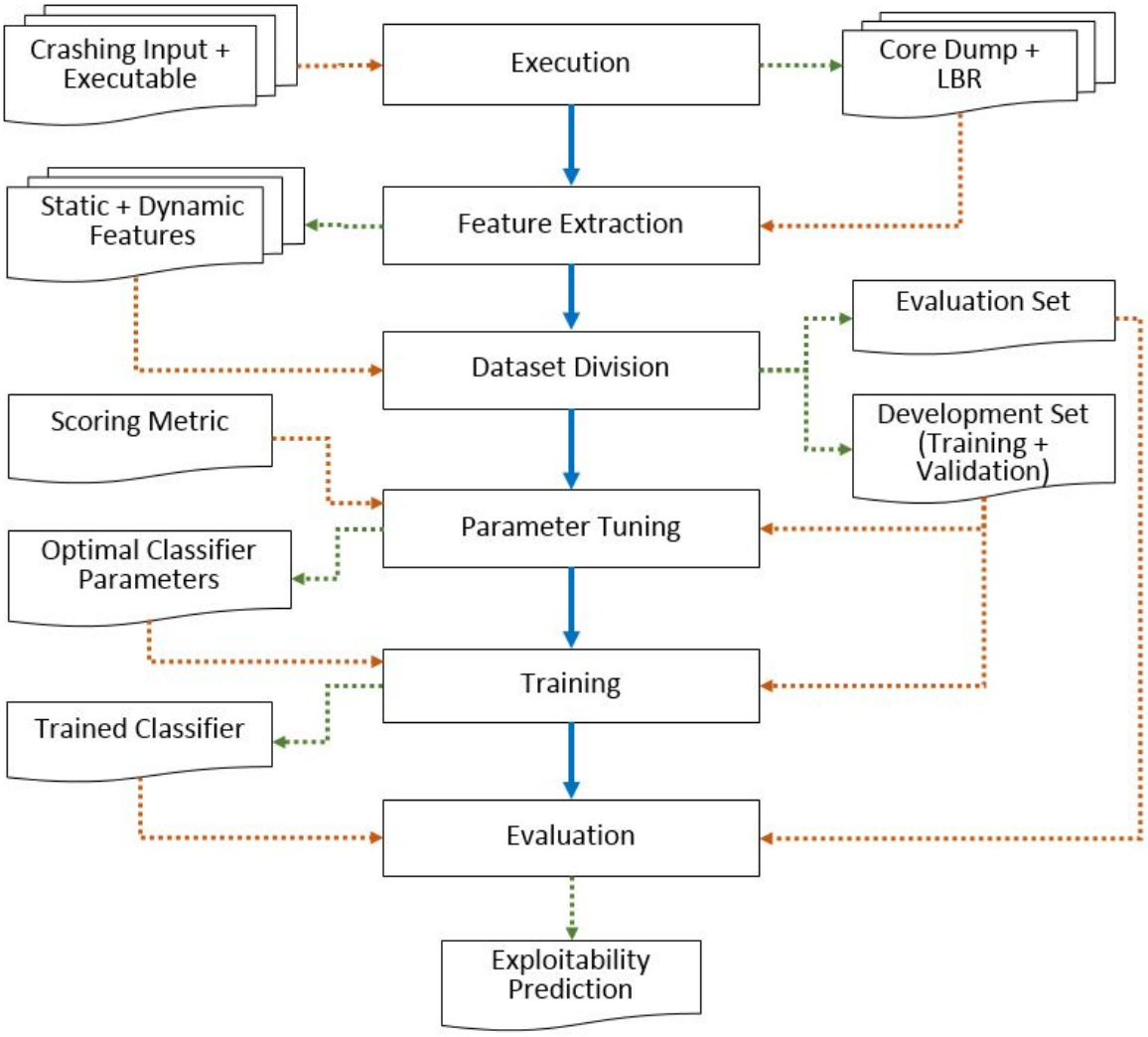

Figure 4.3: Exploitability prediction process

4.2 Exploitability Prediction

We use machine learning based approach to predict exploitability of crashes. Specifically, we use Support Vector Machines (SVMs) as machine learning hypothesis for exploitability prediction and we then extend it for crash prioritization. We also rank static and dynamic features based on their relevance in exploitability prediction. In the following sections, we describe the dataset used for machine learning hypothesis evaluation and its labeling, machine learning model selection and evaluation, crash prioritization and finally feature ranking.

4.2.1 Dataset and Labelling

Machine learning requires large number of samples to train, validate and test a classification algorithm. For our purpose, we have a total of 523 crashes with 166 exploitable and 357 non-exploitable feature vectors. We consider only unique feature vectors because crashes from different applications can have the same feature vector representation. We employ crashes from following sources for our study. We also list the mechanisms used for labelling these data sources.

1. *VDiscovery:* Grieco et al. [14] have packaged the bugs submitted by Mayhem's team [19] in a dataset called *VDiscovery* and have provided the vulnerable binaries in a Linux virtual machine with all the dependencies resolved. *VDiscovery* consists of a total of 402 unique feature vectors.
   Labelling: We utilize the same labelling scheme used by the creators of VDiscovery i.e. explicit consistency checks made using GNU C standard library to indicate memory corruptions of stack and heap and implicit consistency checks due to inconsistent arguments to function calls. Programs which failed these checks are marked exploitable. We have 45 exploitable and 357 non-exploitable unique feature vectors from this dataset.

2. *LAVA:* We have added crashes from dataset created using LAVA [11]. LAVA is a tool that injects synthetic bugs into the programs by identifying locations in execution trace where input bytes are available, minimally modified and do not determine control flow. Modified DTA is used to inject synthetic bugs adhering to the constraints mentioned above and inputs are provided to trigger them. The bugs are memory corruption vulnerabilities and they mimic real world bugs as proven by the authors of LAVA. For our purpose, we used crashes from *toy* program and Linux utilities namely *who* and *uniq*. In total, we have 89 unique feature vectors from this dataset.
   Labelling: Notice that all the samples in this dataset are exploitable by design because the bugs are triggered by the inputs provided i.e. there is a flow from source to sink as discussed in DTA.

3. *Miscellaneous:* We manually downloaded some publically available programs from websites such as ideone [5]. These programs consist of simple C codes. that are available in source code with inputs on which these programs were executed. We compiled them and used only the programs that crashed on our machines. We incorporated a total of 32 unique feature vectors from this dataset.
   Labelling: We utilized a basic DTA implementation to label the crashes. As discussed in DTA section earlier, the reachability of tainted input to the crash site indicates that an attacker may change the control flow of the program leading to exploitation or a denial of service condition. Using this approach we labelled the dataset which consisted of 62 unique feature vectors out of which 32 were exploitable. We incorporated exploitable samples only for classification to reduce class imbalance in the dataset.

4.2.2 Learning Strategy

We pose the exploitability prediction as a binary classification problem with two classes i.e. Exploitable and Non-Exploitable. Each crash is represented as a feature vector in an m-dimensional input space. In our case, $m = 51$. We divide the dataset randomly into two parts - *development* and *evaluation* in the ratio of 3:1, preserving the percentage of each class in the two sets. We further divide the *development* set into *training* and *validation* set and perform cross-validation to determine the best parameters for SVM classifier. We consider the *evaluation* set as the set of unseen samples and use it to evaluate the selected classifier from cross-validation.

We use SVM with *linear* and *rbf* kernels. As discussed in the last section, we have to specify C which is a user-defined regularization parameter for slack variables in the SVM optimization objective (Eq.2.3). In addition, we have to select γ for *rbf* kernel. We set up a grid search with stratified cross-validation to determine the optimal values of these two parameters. Grid search performs exhaustive search over specified parameter values for an estimator (SVM in our case). For *rbf*, we consider all the combinations given by the cartesian product $S_C \times S_\gamma$, where $S_C = \{1, 10^1, 10^2, 10^3\}$ and $S_\gamma = \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1\}$ are the sets of all specified values of C and γ respectively. We use *Stratified K Fold* for cross-validation (with $K = 5$), to preserve the percentage of samples of the two classes in each fold to cater for class imbalance in the dataset. In order to evaluate the models over *validation* set, we use *weighted F1-measure* $F_w \in [0, 1]$ as the scoring metric (Eq.4.1), due to inherent class imbalance.

$$F_w = \frac{1}{\sum_{l \in L} |\hat{y}_l|} \sum_{l \in L} |\hat{y}_l| F(y_l, \hat{y}_l) \tag{4.1}$$

$$F = \frac{2 * P(y, \hat{y}) * R(y, \hat{y})}{P(y, \hat{y}) + R(y, \hat{y})} \tag{4.2}$$

$$P(y, \hat{y}) = \frac{y \cap \hat{y}}{y} \tag{4.3}$$

$$R(y, \hat{y}) = \frac{y \cap \hat{y}}{\hat{y}} \tag{4.4}$$

$$\tag{4.5}$$

where,
y is the set of *predicted* (sample,label) pairs,
$\hat{y}$ is the set of *true* (sample,label) pairs,
L is set of Labels i.e. {Exploitable,Non-Exploitable},
y_l is the subset of y with label l,
P is the *Precision* that specifies the fraction of predicted exploitable that are actually exploitable out of all predicted exploitable cases,
R is the *Recall* that specifies the fraction of predicted exploitable cases that are actually exploitable out of all actually exploitable cases.

Once the user-specified parameters and kernel function are selected, the model is trained on the full development set and evaluated on the evaluation set. We report accuracy in terms of precision, recall and f1-score. We also provide confusion matrix and ROC curve showing relation between True Positive Rate (TPR) and False Positive Rate (FPR) with varying classification threshold.

4.2.3 Crash Prioritization Approach

The output of an SVM classifier as given by Eq.(2.9) is,

$$y_{pred} = sign(W^{*T}\phi(X) + b^*) \tag{4.6}$$

$$y_{pred} = sign(f(x)) \tag{4.7}$$

where, y_{pred} is the prediction in the set $\{-1, +1\}$. However, to prioritize crashes, we need a more continuous prediction that allows us to rank crashes on the basis of exploitability. So, we use probabilistic estimate of exploitability of a crash as a measure for crash prioritization. To convert SVM classification output to probabilistic measure, we utilize Platt Scaling [27] that provides a mapping $\rho : (-\infty, +\infty) \rightarrow [0, 1]$ using a logistic transformation of classifier scores $f(x)$,

$$\rho(y = 1|x) = \frac{1}{1 + \exp(Af(x) + B)} \tag{4.8}$$

where A and B are scalar parameters, estimated using a *maximum likelihood method* that optimizes on the same training & validation set as used by the original classifier f. Although, this results in a slower training process, but it provides an interesting way to rank crashes. For example, a probability prediction of 0.9 for a crash makes it more exploitable and hence more likely to be patched first, than a crash with probability of 0.6. We do not threshold the probability distribution among categories similar to *!exploitable* such as *Probably-Exploitable, Probably-Not-Exploitable* etc. however we propose to use this distribution for assigning priorities to unseen crashes for the purpose of triaging. We also use these probability estimates for generating *ROC* curve over *test* set that helps us to visualize the variation of *TPR* with *FPR* as we vary the threshold. This process for crash prioritization is not used to provide crash exploitability prediction which is purely predicted with an SVM in Eq(4.6), although the two are correlated.

4.2.4 Feature Ranking Approach

Feature ranking provides an indication of the most important features used by a classification model to distinguish between classes. To determine ranks of features used for crash analysis, we perform *recursive feature elimination(RFE)* with a linear SVM over the *development* set. Given an estimator (linear SVM in our case) that assigns weights to features (e.g., the coefficients of the linear model indicated by $Weights_f$), the goal of recursive feature elimination (RFE) is to select features by recursively considering smaller and smaller sets of features (F). First, the estimator is trained on the initial set of

features and weights are assigned to each one of them. Then, features whose absolute weights are the smallest are pruned from the current set of features. This is determined with a parameter $step$ that gives the number of features to prune in each iteration. We have set it to 1. This procedure is recursively repeated on the pruned set until the desired number of features to select ($k = 1$ in our case) is eventually reached. The algorithm used is listed below.

1: **procedure** RFE
2: **Inputs:** $\{clf, k, step, X_{train}, y_{train}\}$
3: **Output:** $\{Rank_f\}$
4: $F \leftarrow \{f_1, f_2, f_3 \dots f_m\}$
5: $Rank_f \leftarrow []$
6: **while** $|F| > k$ **do**
7: $Weights_f \leftarrow$ Train clf on $\{X_{train}, y_{train}\}$ with F features and return coefficients of parameters
8: $Rank_f \leftarrow$ Assign next higher rank to $step$ features with minimum $Weights_f$
9: $F \leftarrow$ Prune $step$ features from F with minimum $Weights_f$.
10: **end while**
11: return $Rank_f$
12: **end procedure**

Chapter 5

Evaluation

5.1 Implementation

The prototype of the proposed system is built on Ubuntu 12.04 LTS operating system supported by Intel x86 core 2 duo 32 bit architecture. The prototype is written in Python and C++. The *static features* are extracted from *core dump* using GDB [2] which is an open source debugger provided by the GNU foundation. We have utilized gdb-python-api framework provided by GDB that supports custom user commands. We have used capstone [1], which is a lightweight disassembly framework, for disassembling the crashing instruction and branch records available from LBR. We have also made use of readelf [6] utility to extract the permissions of allocated memory segments. We use a PIN [22] based simulator to simulate LBR tracing mechanism. Since the dataset used for training is available within a virtual machine that does not support LBR, we simulated LBR extraction using PIN by tracking every branch instruction with source and destination addresses. PIN based simulations have been used in previous studies such as Intel Processor Trace simulation in Gist [18] and do not affect the results in any way except for increase in runtime overhead. In addition to LBR simulation, we also use PIN for taint tracking for the purpose of DTA. We used a simple taint engine based on Salwan's work [8] and modified it suitably to run on 32-bit architecture that we have currently. We do not make any assumptions about Intel based architecture used for LBR simulation or DTA and the approach will work with 64-bit computers as well. We have used sklearn [7], a robust machine learning library available in python, for classification, crash prioritization and feature ranking.

5.2 Results and Discussion

5.2.1 Parameter tuning and model selection

As discussed in the last section, we use an SVM classification model with two kernels *linear* and *rbf*. The user-defined parameters are C and γ and the scoring metric for cross-validation is *weighted f1-measure*. Table 5.1 gives the scoring of SVM models with different parameters from the ordered set

$S_C \times S_\gamma$. Based on the scores we find that the best SVM model with *rbf* kernel and parameters $C = 100$ and $\gamma = 0.001$ (highlighted in table).

Table 5.1: Cross-Validation Scores for SVM models

Kernel	C	γ	Weighted f1-Score
rbf	1	0.0001	0.547 (+/-0.006)
rbf	1	0.001	0.547 (+/-0.006)
rbf	1	0.01	0.848 (+/-0.028)
rbf	1	0.1	0.868 (+/-0.047)
rbf	1	1	0.740 (+/-0.079)
rbf	10	0.0001	0.554 (+/-0.029)
rbf	10	0.001	0.847 (+/-0.101)
rbf	10	0.01	0.873 (+/-0.063)
rbf	10	0.1	0.871 (+/-0.051)
rbf	10	1	0.771 (+/-0.079)
rbf	100	0.0001	0.831 (+/-0.080)
rbf	**100**	**0.001**	**0.880 (+/-0.058)**
rbf	100	0.01	0.863 (+/-0.075)
rbf	100	0.1	0.797 (+/-0.043)
rbf	100	1	0.771 (+/-0.079)
rbf	1000	0.0001	0.874 (+/-0.064)
rbf	1000	0.001	0.875 (+/-0.068)
rbf	1000	0.01	0.837 (+/-0.088)
rbf	1000	0.1	0.800 (+/-0.052)
rbf	1000	1	0.771 (+/-0.079)
linear	1	-	0.871 (+/-0.060)
linear	10	-	0.859 (+/-0.048)
linear	100	-	0.859 (+/-0.048)
linear	1000	-	0.859 (+/-0.048)

It is interesting to note here that when C is small and γ is also small, the f1-score is low. For example, if we take $C = 1$ and $\gamma = 0.0001$, the score is only 0.547. This case clearly indicates under-fitting as we also discussed in earlier section. If we consider a high C and high γ values, such as 1000 and 1 respectively, we find that the score is again low, i.e. 0.771. This scenario indicates over-fitting. If we consider a fixed C (say $C = 1$), then we observe that with increasing γ, the score first increases and then decreases. This shows a trend from under-fitting of the training set to over-fitting with change in γ. Therefore, we used grid search with cross-validation to determine the optimal parameter values for the given *development* dataset.

Finally, we also note that the scores for linear SVM are consistent apart from a minor drop due to an increase in value of C from 1 to 10. Although this does not represent a linearly separable case (as

f1-score is not 1), but it does indicate presence of errors in the training data because some of the samples are always classified incorrectly even if the model tries to overfit the data by increasing value of C. Possible reasons for erroneous data include dataset labeling procedure, distinguishing feature information and loss of exploitability related information in core-dump and LBR.

5.2.2 Exploitability Prediction

We now train the selected classifier on the *full development* dataset that includes validation set and evaluate it on *evaluation* set. We report precision, recall and f1-score for two classes and accuracy for both classes in Table 5.2. Accuracy is the ratio of total number of correctly classified samples to the total number of samples. This error metric is inaccurate for imbalanced datasets because it will be biased towards the class with higher number of samples. Support represents the total number of evaluation samples available in a particular class.

We also list the confusion matrix in Table 5.3, that numerically describes the classifier prediction with actual class of the samples.

Table 5.2: Exploitability Prediction Accuracy

class	precision	recall	f1-score	accuracy	support
Exp.	0.95	0.81	0.88	-	48
Non-Exp.	0.92	0.98	0.95	-	109
	-	-	-	0.93	157

Table 5.3: Confusion Matrix

Predicted/Actual	Exploitable	Non-Exploitable
Exploitable	39	9
Non-Exploitable	2	107

Figure 5.1 shows the Receiver-Operating-Characteristic (ROC) curve for the SVM model selected using grid-search over evaluation set. This curve shows the relation between *TPR* and *FPR* as the threshold varies. The threshold is a probability $P_r \in [0, 1]$ such that if $\rho(y = 1|x) \geq P_r$ the feature vector x is predicted as *Exploitable*, else it is predicted as *Non-Exploitable*. ρ transforms the classifier scores to probabilistic estimates using logistic regression as discussed in the last section on *crash prioritization* (see Eq.4.8).

Table 5.4 shows time taken by exploitability prediction tasks averaged over 100 real-world applications taken from VDiscovery dataset. We found SVM model training time for 300 testcases to be 0.451

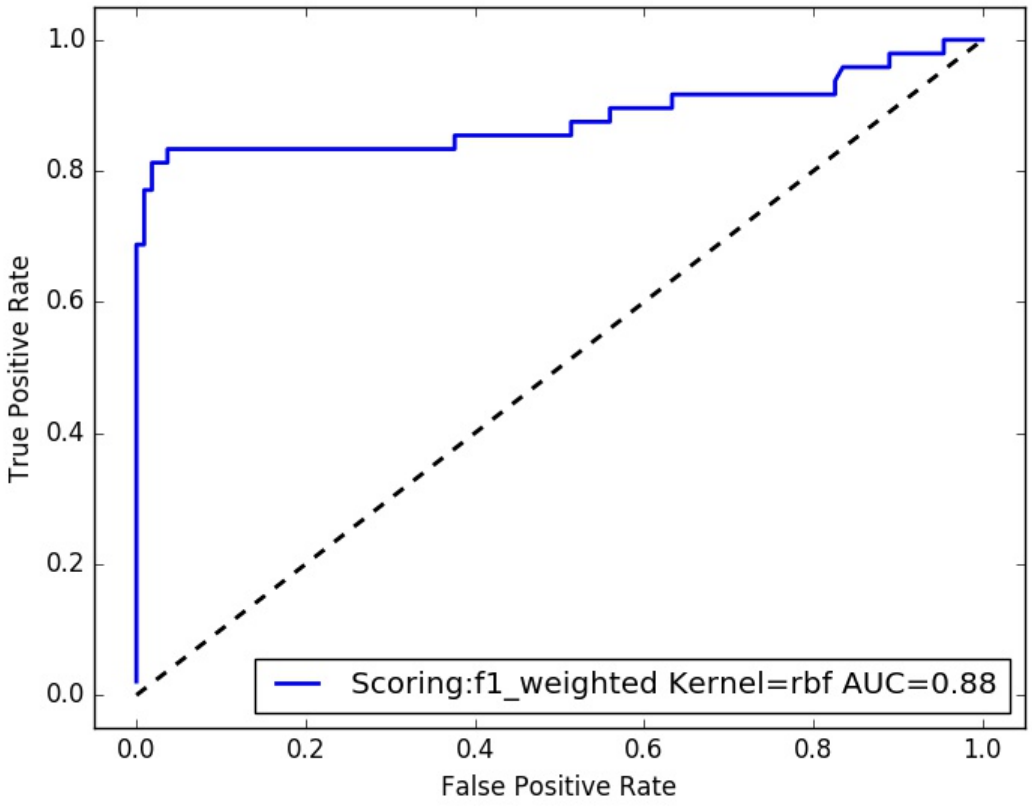

Figure 5.1: ROC curve on evaluation set with selected classifier.

Table 5.4: Average time statistics for exploitability prediction tasks

Task	Execution Time per testcase (sec)
Original execution	0.401
LBR simulation with PIN	4.522
Dumping of core	0.426
Feature Extraction (static+dyn)	1.356
Dynamic Taint Analysis with PIN	73.493

sec excluding optimal parameter determination and exploitability prediction time for 150 testcases to be 0.00423 sec. This clearly indicates feasibility of Exniffer, for exploitability prediction in terms of efficiency in real-time production systems. We can also see that DTA with PIN introduces a lot of overhead as compared to Exniffer and is unsuitable for production systems. It is also worth noting that hardware assisted LBR extraction will be faster than LBR simulation which currently takes around 4.5 sec.

5.2.3 Crash Prioritization

As discussed in the last section, we can utilize the probability assigned for a test sample as a measure of its priority to fix the bug. Consider the Listing 5.1 below which is part of evaluation set. Perhaps the coder is interested to convert upper case letters in an array $temp$ to lower case. But in the process he forgot to copy the values from the input array b to $temp$. And finally, he copies $temp$ to a[i]. Due to infinite loop at line 6, i goes out of bounds of array a and results in a segmentation fault at line 12.

Listing 5.1: Prioritizing a non-exploitable crash

```
1  int main() {
2          char a[40][40];
3          long int b[40];
4          int i=0;
5          char temp[40];
6          while(1) {    // infinite loop
7                  long int j;
8                  for(j=0;j<=strlen(temp);j++) {
9                          if(temp[j]>=65 && temp[j]<=90)
10                                 temp[j]+=32;
11                 }
12                 strcpy(a[i],temp);  /* temp is copied in a[i] -> Crash! */
13                 scanf("%ld",&b[i]); /* user input in b[i] */
14                 i++;
```

```
15          }
16
17          long int j;
18          for(j=0;j<i;j++) {
19                  printf("%s %ld ",a[j],b[j]);
20          }
21          return 0;
22  }
```

Although naively written, this case is interesting to consider because it is *non-exploitable*. The input of attacker does not reach the crash site because input in b is totally disjoint from array a. We also used *DTA* to confirm that crash instruction is not affected by user input. However, this case is predicted exploitable by both !exploitable and Exniffer.

!exploitable classifies this case at the highest level of exploitability and provides the following explanation:
The target crashed on an access violation at an address matching the destination operand of the instruction. This likely indicates a write access violation, which means the attacker may control the write address and/or value.

The crashing instruction is *movl %eax, (%edx)* which writes the value of in register *%eax* to memory location at address *%edx*. So, there is a write access violation but as we already know it is not exploitable.

Exniffer also predicted this case to be exploitable. However, if we rank the crashes based on probability estimates, we found that this case had a probability of *0.573* only which is not that high and hence does not represent a severely exploitable case.

In another example from evaluation set, we consider an *exploitable* scenario and report probability estimate of the classifier. We also compare it with *!exploitable*. In the Listing 5.2 below, we see that the user's input is stored in $input$ array allocated on heap. Another array B is updated using *update* method, however this results in a segmentation fault at line 8. The parameter *row* of *update* function receives the value 30 equal to *len1*. The loop at line 5 increments i to 30. The maximum number of rows in B are 30 and it can be indexed until 29, however, in line 8 when the coder tries to write the value of $input[j]$ at $B[i]$, where $i = 30$, it results in crash. The coder should have re-initialized variable i. It is very obvious from the crashing instruction that this is an exploitable scenario since the value of $input$ array directly affects the crash site. We also confirmed exploitability of this crash with DTA.

Exniffer predicted this case as *Exploitable*. We found the probability estimate of this scenario is 0.923 which gives a fairly high confidence to keep the priority of fixing this crash higher.

!exploitable on the other hand predicted that the crash is *Probably_Exploitable*, which is a lower category for a clearly exploitable case. It is also clear with the following explanation that !exploitable does not enough information to make the decision:
The target crashed on an access violation at an address matching the destination operand of the instruction. This likely indicates a write access violation, which means the attacker may control write address and/or value. However, there is a chance it could be a NULL dereference.

Listing 5.2: Prioritizing an exploitable crash

```
1  void update(int **B,int row,int col,int *input,int k){
2          printf("Update\n");
3          int i,j;
4          i=k;
5          for(i=0;i<row;i++)
6                  printf("\n%d %d",i,input[i]);
7          for(j=0;j<2;j++)
8                  B[i][j]=input[j];      // crash! coder forgot to re-initialize i
9  }
10
11 int main()
12 {
13         int len1,len2,i,k=0,testcase,l=0;
14         int *input = (int *)malloc(sizeof(int)*3);
15
16         len2 = 3;
17         len1 = 30;
18         int **B = (int **)malloc((len1) * sizeof(int *));
19         for (i=0; i<(len1); i++)
20                 B[i] = (int *)malloc((len2) * sizeof(int));
21
22         printf("Enter no. of test cases\n");
23         scanf("%d",&testcase);
24         while(l<testcase){
25                 //      printf("Enter array : ");
26                 for(i=0;i<3;i++){
```

```
27                          scanf("%d",&input[i]);
28                          printf("\n%d %d",i,input[i]); // user input
29                  }
30
31                  if(unique(B,len1,len2,input,k)){   // unique returns true
32                          printf("%d %d %d\n",len1,len2,k);
33                          // control redirected to update
34                          update(B,len1,len2,input,k);
35                          k++;
36                  }
37                  l++;
38          }
39          len2=3;
40          len1=k;
41          print_array(B,len1,len2);
42          return 0;
43  }
```

5.2.4 Feature Ranking

As discussed earlier, we use Recursive Feature Elimination (RFE) to rank the features with a linear SVM. Figure 5.5 shows top ten features available as a result of application of RFE to training set.

Table 5.5: Top 10 features based on RFE

id	Feature Description
1.	Backtrace is Corrupt
13.	Memory operand is Null
8.	EIP Segment is Executable
19.	Operand is immediate
11.	Memory operand is Source
42.	Access Voilation Signal
49.	Loop
17.	#Operands = 3+
46.	Conditional Jump
43.	Floating point exception Signal

The top feature i.e. corruption of backtrace is generally a result of buffer overflow that overwrites the saved *base pointer register (EBP)* of the last frame which leads to incorrect unwinding of stack frames. It is therefore a major feature to determine exploitable conditions and is correctly predicted by RFE algorithm.

To accomplish Data Execution Prevention (DEP),stack generally has write permissions but not execute. *EIP* must ideally point to the segments that only have execute permissions, but if the instruction register points to a write segment without execute permission, then it will result in a crash. The feature *EIP Segment is Executable* in Table 5.5 highlights this.

The feature *Loop*, extracted from LBR, represents if the crashing instruction is within a loop. It is a known fact that overflow of neighboring values outside of array bounds generally happens within a loop such as copying operation using *strcpy*. Another feature *Conditional Jump* extracted from LBR checks if the last branch instruction was a conditional jump. Conditional jump has a condition that may be affected by the attacker's input. The branch instruction may also coincide with crashing instruction which is a more serious threat as it may give the control flow in the hands of the attacker.

In Table 5.5, we also find features that support non-exploitable scenarios, such as *Memory operand is null* which represent a *null* deference. Also, *Floating point exception signal* generally represents a non-exploitable case of *division by zero*. It may also represent cases which are exploitable to the extent of causing *DoS* i.e. an application crash.

We also note that features that do not clearly represent either of the classes are also in the top-ten list such as *Operand is immediate* and *#Operands = 3+*. While the former feature represents the availability of *immediate* operand in the crashing instruction that gives an indication of infeasibility of manipulation of register values, the latter checks if there are three or more operands in the crashing instruction. *Memory operand is Source* is also a distinguishing feature that adds information on read-access violation. A read-access violation is caused due to an illegal memory read i.e. either the memory address of that segment does not have read permission or the memory segment is not allocated at all. This may represent exploitable cases if the register that is dereferenced is tainted.

5.2.5 Comparison with !exploitable

!exploitable is rule based classifier, so it does not have any hypothesis to learn. It can be used directly on the entire dataset. Table 5.6 gives the prediction of *!exploitable* on the entire dataset using a confusion matrix represented as percentage for respective classes. The first column is divided as per the categories assigned by !exploitable. Predicted categories are given in rows while the actual categories are listed in columns.

Out of 100% of the *exploitable* cases, we found that 15% of all the exploitable samples and 7% of *non-exploitable* samples are marked as *exploitable*. A large number of *exploitable* samples are labelled *unknown* because there was not enough information for *!exploitable* to label the crash in one of the other categories. Out of 100% of non-exploitable cases, 59% are marked *Probably-Not-Exploitable* which shows that *!exploitable* performs better on *non-exploitable* cases as compared to *exploitable* scenarios.

Table 5.6: Confusion matrix for !exploitable

Predicted/Actual	Exploitable	Non-Exploitable
Exploitable	15%	7%
Probably-Exploitable	2%	18%
Probably-Not-Exploitable	1%	59%
Unknown	82%	16%

However, the results show that accuracy of *!exploitable* is low as compared to Exniffer primarily because it consists of a rule-based engine i.e. the features as well as decision functions are hard-coded, which makes it difficult for the system to adapt to different scenarios. Exniffer on the other hand uses machine learning that automatically learns relevant features and decision boundary given sufficient data thereby adapting and generalizing to a range of different scenarios. Using the crash prioritization approach, we also provide a continuous crash prioritizing technique for crash ranking. Each crash is associated with a probabilistic measure of its exploitability. However, in case of *!exploitable*, if 15% of the samples are *exploitable*, then it does not answer the question that among these samples, which ones are to be fixed first i.e. there is no sub-class priorities available. Table 5.7 shows the exact improvement of Exniffer over *!exploitable*. We found that the recall for Exploitable class is very low i.e. 0.16 as compared to Exniffer.

Table 5.7: Exploitability Prediction Accuracy

class	precision	recall	f1-score
Exp.	0.62	0.16	0.26
Non-Exp.	0.26	0.74	0.38

Finally, in *!exploitable* all the features are hard-coded and the importance of features is pre-decided. Since classification process is not data-driven, it is difficult to create complex decision functions using these features to classify a crash accurately.

Chapter 6

Conclusion and Future Work

Today there are millions of bugs reported across thousands of software products for example 1.7 million bugs are reported in 12,275 projects at Launchpad since 2004 [3]. Bugs resulting in these crashes can be exploitable and must be fixed in timely manner to avoid security ramifications. As a result, organizations are moving towards automated crash analysis to save time, effort and resources. Since there is enough crash data available to a general organization, it is possible to perform data-driven automated crash analysis that can help developers to identify and patch security crashes first.

In this paper, we proposed Exniffer that uses machine learning to classify and prioritize crashes. We used a novel combination of static and dynamic features extracted from core-dump and hardware branch tracing facilities that incurs negligible runtime-overhead which makes it suitable for production systems. The results show that the approach is effective in exploitability prediction and crash prioritization. With this system, we were able to achieve a true positive rate of 81% for exploitable scenarios and a very low false positive rate of 2%. We also identified relevant features for exploitability prediction and found that both static and dynamic features play a role in separating exploitable and non-exploitable cases.

In future, we plan to extend our work by incorporating the following:

1. Expand feature set by engineering of static and dynamic features to improve accuracy further.
2. Increase dataset size to perform a more exhaustive testing of the system.
3. Consider memory vulnerabilities from executables compiled from sources other than C and C++.
4. Predict vulnerability type for an exploitable case.
5. Intel PT allows capturing branch outcomes and timing information with low overhead ($\leq 5\%$). Leveraging the fact that short term memory for failure diagnosis is sufficient to determine root cause, we will try to reduce the overhead even further.

Related Publications

"*Exniffer: Learning to Rank Crashes by Assessing the Exploitability from Memory Dump*" Shubham Tripathi, Gustavo Grieco, Sanjay Rawat. *in submission* of **24th Asia-Pacific Software Engineering Conference, 2017**, Nanjing, China.

Bibliography

[1] Capstone: The ultimate disassembler. http://www.capstone-engine.org/.

[2] Gdb: The gnu project debugger. https://www.sourceware.org/gdb/.

[3] Launchpad software collaboration platform bug tracker statistics. https://launchpad.net/bugs.

[4] Nehalem core pmu. https://software.intel.com/en-us/search/gss/nehalem.

[5] Online compiler and debugging tool. http://ideone.com/recent.

[6] Readelf: Gnu binary utilities. https://sourceware.org/binutils/docs/binutils/readelf.html.

[7] scikit-learn: Machine learning in python. http://scikit-learn.org/stable/.

[8] Taint analysis pin tools. https://github.com/JonathanSalwan/PinTools.

[9] !exploitable, 2013. https://msecdbg.codeplex.com/.

[10] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. Retracer: triaging crashes by reverse execution from partial memory dumps. In *Proc. of 38th International Conference on Software Engineering*, pages 820–831, New York, USA, 2016. ACM.

[11] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, F. Robertson, Wil; Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *Proc. of IEEE Symposium on Security and Privacy (SP)*, pages 110 – 121, 2016.

[12] K. J. Eom, J. Y. Paik, S. K. Mok, H. G. Jeon, E. S. Cho, D. W. Kim, and J. Ryu. Automated crash filtering for arm binary programs. In *Proc. of Computer Software and Applications Conference*, pages 478 – 483. IEEE, 2015.

[13] J. Foote. Cert triage tools. https://www.cert.org/vulnerability-analysis/tools/triage.cfm?

[14] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proc. of Conference on Data and Application Security and Privacy*, pages 85–96, NY, USA, 2016. ACM.

[15] S. Heelan and D. Kroening. Automatic generation of control flow hijacking exploits for software vulnerabilities. In *Univ. Oxford*, London, U.K., 2009.

[16] S.-K. Huang, M.-H. Huang, P.-Y. Huang, H.-L. Lu, and C.-W. Lai. Software crash analysis for automatic exploit generation on binary programs. In *Proc. of IEEE Transactions on Reliability*, pages 270 – 289, 2014.

[17] S. L. Joy Arulraj, Guoliang Jin. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *Proc. of international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 207–222, NY, USA, 2014.

[18] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in production failures. In *Proc. of Symposium on Operating Systems Principles (SOSP)*, pages 344–360, NY, USA, 2015. ACM.

[19] S. Kil-Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proc. of IEEE Symposium on Security and Privacy*, pages 380 – 394, 2012.

[20] D. Kim, X. Wang, S. Kim, A. Zeller, S. C. Cheung, and S. Park. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. In *Proc. of IEEE Transactions on Software Engineering*, pages 430 – 447, 2011.

[21] G. Y. J. L. Z. S. Y. Kucuk. Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability. In *Proc. of 1st IEEE Symposium on Privacy-Aware Computing*, 2017.

[22] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[23] C. Miller, J. Caballero, N. M. Johnson, M. Gyung Kang, S. McCamant, P. Poosankam, and D. Song. Crash analysis with bitblaze, 2010.

[24] S. Rawat and L. Mounier. Finding buffer overflow inducing loops in binary executables. In *IEEE Sixth International Conference on Software Security and Reliability*, pages 177 – 186, 2012.

[25] M. B. V.-T. P. M.-D. N. A. Roychoudhury. Directed greybox fuzzing. In *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344. ACM, 2017.

[26] M. D. F. L. G. Z. V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298. ACM, 2017.

[27] H.-T. L. C.-J. L. R. C. Weng. A note on platts probabilistic outputs for support vector machines. In *Machine Learning*, pages 68:267–276, 2007.

[28] Z. P. W. J. W. X. Z. Wu. Program crash analysis based on taint analysis. In *Proc. of Ninth IEEE International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 492 – 498, 2014.

[29] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proc.CCS'16*, pages 529–540, New York, NY, USA, 2016. ACM.

[30] T. F. Yi, C. Feng, and C. J. Tang. Binary vulnerability exploitability analysis. In *Proc. of International Conference on Information System and Artificial Intelligence (ISAI)*, pages 181–185. IEEE, 2016.