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

Shubham Tripathi
International Institute of Information Technology
Hyderabad, India
email: shubham.t@research.iiit.ac.in

Gustavo Grieco
CIFASIS-CONICET
Rosario, Argentina
email: gg@cifasis-conicet.gov.ar

Sanjay Rawat
Vrije Universiteit
Amsterdam, NL
email: s.rawat@vu.nl

***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.

I. Introduction

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. A crash as a result of safety critical bug may not necessarily be exploitable as it may not depend on the malicious inputs, whereas a crash due to a security bug necessarily depend on the malicious inputs [1]. 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 [2], we will notice that on 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.

```
 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     ch=getchar();
 7     name[i]=ch;  // crash!
 8     i++;
 9   }
10   name[i]='\0';  // inserting null character at end
11   printf("Name: %s",name);
12   return 0;
13 }
```

Listing 1. Exploitable Example

Listing 1 shows a classic buffer overflow example that crashes at line 9. The code accepts user input characters in a buffer until a newline and store it in a fixed size buffer *name* of maximum 30 characters. A string with length sufficiently larger than 30 characters will lead to corruption of the memory address containing the return address of the last function on the stack and when the current function returns, the corrupt value in return address will lead into a segmentation fault. An attacker with a crafted input string can lead 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.

```
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. Non-Exploitable Example

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.

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. Although information available from core-dump is vital, it may not always be sufficient for crash analysis. Tools such as Crashfilter [3], ExpTracer [4] and Bitblaze [5] employ static and dynamic analysis to enrich the information available from a core-dump for a more accurate analysis. However, such types of analysis have following limitations:

978-1-5386-3681-7/17 $31.00 © 2017 IEEE
DOI 10.1109/APSEC.2017.30

1) Computational cost: Dynamic analysis (Dynamic Taint Analysis (DTA), in particular) have significant runtime overhead [6] [7] which makes them not suitable for real-time monitoring and analysis of production systems.
2) Bug Reproducibility: More expensive analysis, like DTA, can be applied offline, but running the application with the crash producing inputs. This, however, will meet the well known problem of non-reproducibility of the bug [8] [9] as the analysis environment is different from the production environment in which the bug was detected.

To counter the limitations discussed above, we propose a mechanism for *online analysis* that is computationally inexpensive and uses real-time production environment for exploitability prediction without re-running the application over crashed input. In addition to statically analyzing core-dump, we leverage hardware debugging extensions specifically *Last Branch Record (LBR)* available in recent processors for dynamic analysis to reduce runtime overhead. LBR has been shown to be effective for root-cause analysis in previous studies [10] [6], for example, LBR saves a set of few (generally 16) most recent branches and has been used to trace program flows and determine code coverage.

Rule based engines such as *!exploitable* [11] predicts exploitability of a crash by analyzing the crash state of a program within a debugger. Although *!exploitable* has less runtime overhead, it fails to *generalize* to different crash scenarios and leads to many false alarms, for example, *!exploitable* incorrectly predicts Listing 2 as exploitable. So, instead of specialized exploitability rules developed manually, we propose to utilize machine learning to automatically determine generalized rules for exploitability prediction.

In this work, 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: exploitable and non-exploitable, Exniffer learns an SVM hypothesis which can be used to classify an unseen sample. We utilize publicly available dataset: VDiscovery, LAVA and some additional C/C++ test cases 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 online and real-time analysis in production systems. We only assume availability of a binary executable (stripped), 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 exploitability for a large number of crashes, we assign a priority to each case that estimates how severe the crash is with respect to exploitability. This allows the developers to fix the most relevant crashes first. So, we extend Exniffer 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.

Specifically, we make following contributions:

1) Large scale experiment setup for exploitability prediction using machine learning.
2) Design of lightweight static and dynamic features that represents a crash extracted from core-dump and LBR.
3) Crash prioritization to determine exploitability rating of a crash.
4) Ranking of features to provide insight into the classification algorithm.
5) Implementation and comparison of Exniffer with !exploitable.

We shall describe the proposed technique in Section II that describes the background, feature design and exploitability prediction algorithm. We then evaluate, discuss and compare Exniffer in Section III. Next we list the related work done in the field in Section IV and conclude in Section V.

II. Proposed Technique

A. Background

1) 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. A core-dump also contains information about processor registers, such as program counter and stack pointer. It also contains memory management information such as memory segment size, addresses and permissions. A crash can be dumped as a kernel-dump that contains memory state of kernel or a mini-dump 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 stripped ELF mini-dumps.

2) Last Branch Record (LBR): LBR is a hardware debugging utility for branch tracing provided by Intel as part of processor's performance monitoring unit. When enabled, LBR stores the last few branches in the form of source and destination addresses using a circular ring of hardware registers and incurs almost zero overhead. The number of records that LBR can store vary with the model of the processor (4 records in Pentium 4 to 16 in Nehalem) and are therefore called Model Specific Registers (MSR) [12]. LBR can also be configured to record different types of branch instructions, for example conditional branches, unconditional jumps, calls, returns etc. For our purpose, we use LBR to trace 16 branch instructions consisting of source and destination addresses.

3) Dynamic Taint Analysis: Dynamic Taint Analysis is a form of data flow analysis that tracks data as it flows through a program while it is executed [13]. 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 memory corruption bug. 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 labelling the dataset in one of the two classes: Exploitable and Non-Exploitable, instead of exploitability prediction.

DTA works at the binary level and hence it can perform taint analysis on loaded 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. The tracked tainted information 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:

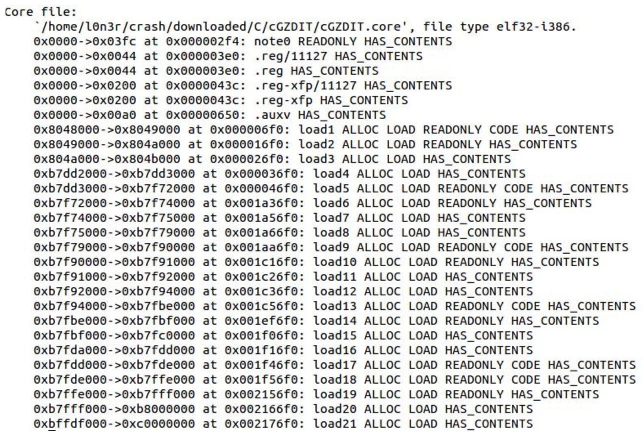
1) *Taint Initialization:* The input to the program controlled by a potential attacker is marked as tainted: system calls such as read are hooked and memory locations where the input data flows are captured and stored.
2) *Taint Propagation:* Propagation of taint can occur in three ways. First, a memory address containing the tainted data is read using a load instruction such as `mov [%eax],%ebx`. Second, memory write instructions such as `mov %eax,[%esp]`, results in tainting of memory address if the register to be written is tainted. 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.

B. Feature Design

Machine learning requires crashes to be represented in feature vectors. These features have to be designed and then extracted from core-dump and LBR. 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 I lists all the features that describe a feature vector.

a) Static Features: Static features are lightweight features extracted from a core-dump. They can be grouped in the following categories:

1) *Stack Unwinding*: Unwinding of stack, traces back functions that were active at the time of crash. This is popularly known as *backtrace* in debugging terminology. In x86 architecture, the base pointer register points to the location in memory that contains the address of base pointer of the last function or the caller of current function. 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`. 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 stack corruption.
2) *Special Registers*: x86 architecture provides special registers in the form of instruction pointer, base pointer



```
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
```

Fig. 1. Valid memory addresses: Core file contents displayed using GDB shows the address range of loaded sections at the time of crash

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 in knowing if such registers point to valid memory locations or not, where valid memory locations are those segments that are allocated for the process by the operating system. 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*. For example, segments containing stack must only have 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 1 shows the typical section information available in a core-dump file extracted using GDB [14]. First and second columns provide the valid address range for each section at the time of crash. For example, `[0xbffdf000, 0xc0000000)` is the address range of the last section which is generally part of stack. In this example, the program crashed at a `mov` instruction:

```
(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, i.e. `0xc0000000`. But since, this is an out of bounds memory access as shown in Figure 1, the program crashes as a result of segmentation fault. Also note that the sections are synonymous with segments in this case, however, in general there can be multiple sections in a segment.
3) *Crashing Instruction*: The instruction at which the program has crashed contains important information regarding the type of access violation. A memory read instruc-

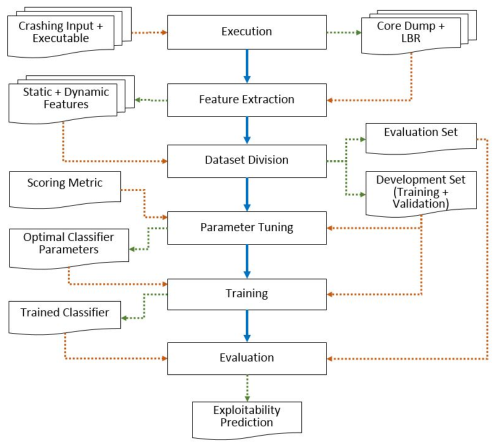
tion represents read access violation while a memory write instruction represents write access violation. We include the type of access violation in the feature set. We also check whether the operands are contained within the memory segments allocated at the time of crash. In addition, whether the type of instruction is branch or not is important to capture change in control flow 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 access violation signal is sent to a process when it makes an invalid virtual memory reference resulting in segmentation fault. 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.

b) 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 [6] 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) *Occurrence of crash in a loop*: Vulnerabilities such as stack and heap overflow have a high chance of occurring inside a loop where memory is sequentially written [15]. 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.



Crashing Input + Executable
Execution
Core Dump + LBR
Static + Dynamic Features
Feature Extraction
Evaluation Set
Dataset Division
Scoring Metric
Development Set (Training + Validation)
Optimal Classifier Parameters
Parameter Tuning
Training
Trained Classifier
Evaluation
Exploitability Prediction

Fig. 2. Exploitability prediction process

C. Exploitability Prediction

We use Support Vector Machines (SVM) as machine learning hypothesis for exploitability prediction. We 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, labeling procedure, SVM model selection and evaluation, crash prioritization and feature ranking.

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 samples from three different sources: VDiscovery, LAVA and Miscellaneous. We consider only unique samples because crashes from different applications can have the same feature vector representation.

TABLE I
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 Violation 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

Labeling mechanism: We employ one of the following two mechanisms to label datasets from different sources:

- *Taintflow analysis:* As discussed in section II-A3, DTA can be used to determine exploitability of a crash by tracing input to the crash site. We employ DTA for Miscellaneous test cases.
- *VDiscovery labelling mechanism:* VDiscovery is a publically available dataset of bugs submitted by Mayhem's team [16] and packaged by Grieco et al. [17]. VDiscover is a machine learning based tool that extracts features from VDiscovery dataset and predicts if a test case is likely to contain a software vulnerability (Please note the difference in the names of tool and dataset). The creators of VDiscovery dataset have used a manual ad-hoc labelling mechanism to label this dataset on which the tool VDiscover is trained and evaluated. For labelling VDiscovery, the creators have used 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. For our purpose, we use VDiscovery dataset labelled with this mechanism.
  Notice the fact that the application (i.e. test case) to be labelled has to be compiled with special compiler flags such as stack protector, heap consistency checks etc.

The above mechanisms used for labelling the dataset cannot be used for exploitability prediction because they are computationally expensive and are not suitable for online analysis. In addition, we do not assume availability of binaries compiled with special flags for exploitability prediction.

We employ crashes from following sources for our study:

- *VDiscovery:* Grieco et al. [17] have packaged the vulnerable binaries in a Linux virtual machine with all the dependencies resolved. This dataset consists of a total of 402 unique samples.
  Labelling: We utilize the same labelling scheme used by the creators of VDiscovery as discussed above We have 45 exploitable and 357 non-exploitable unique samples from this dataset.
- *LAVA:* We have added crashes from dataset created using LAVA [18]. LAVA is a tool that injects synthetic memory corruption bugs into the programs using a modified DTA such that they mimic real world bugs. For our purpose, we used crashes from *toy* program and Linux utilities namely *who* and *uniq*. In total, we have 89 unique samples from this dataset.
  Labelling: All the samples in this dataset are exploitable by design because the bugs are triggered by the synthetic inputs.
- *Miscellaneous:* We manually downloaded some publically available programs from websites such as ideone [19]. These programs consist of simple C codes. We compiled them and used only the programs that crashed on our machines. We incorporated a total of 32 unique samples from this dataset.
  Labelling: We utilized a basic DTA implementation to label the crashes. The dataset consists of 62 unique samples out of which 32 were found to be exploitable. We incorporated exploitable samples only for classification to reduce class imbalance in the dataset.

2) Machine learning hypothesis: 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 use Support Vector Machine (SVM) to learn an optimal separating hyperplane that maximizes separation between two classes. SVM can also learn non-linear discriminant functions by effectively mapping original input space to high-dimensional feature space.

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

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

and $\epsilon_i \geq 0$, where $\epsilon_i \in \mathbb{R}$ 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 with optimization objective as minimization of,

$$\frac{1}{2} W^T W + C \sum_{i=1}^{n} \epsilon_i \tag{2}$$

Here C is the user defined regularization parameter that penalizes the slack variables and hence controls overfitting and underfitting of data by SVM.

The solution to the optimization objective results in the optimum hyperplane (W^*, b^*) . The class of a test sample X can now be predicted as,

$$sign(W^{*T} \phi(X) + b^*) \tag{3}$$

$$sign(\sum_{i=1}^{n} \mu_i^* y_i \phi(X_i)^T \phi(X) + b^*) \tag{4}$$

where μ_i's are positive real numbers that maximize the *dual* of the optimization objective in Eq.2.

Using *kernel functions* ($K(X_i, X_j) = \phi(X_i)^T \phi(X_j)$), it is possible for SVM to learn and also use maximum margin hyperplane efficiently in $\Re$ without computing the mapping explicitly [20]. A linear kernel is defined as $K(X_i, X_j) = X_i^T X_j$ while an rbf kernel is a 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.

3) 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 to be minimized, the term $\sum_{i=1}^{n} \epsilon_i$ has to very small, which means that the selected model makes little or no errors in classification and hence overfits the data. On the other hand, if C is very small, then ϵ_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.

4) Learning Strategy: 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). 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. In order to evaluate the models over *validation* set, we use *weighted F1-measure* as the scoring metric due to inherent class imbalance. 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.

5) Crash Prioritization Approach: The output of an SVM classifier as given by Eq.(4) is,

$$y_{pred} = sign(W^{*T} \phi(X) + b^*) \tag{5}$$

$$y_{pred} = sign(f(x)) \tag{6}$$

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 [21] that provides a mapping $\rho : (-\infty, +\infty) \rightarrow [0, 1]$ using a logistic transformation of classifier scores $f(x)$,

$$\rho(y = 1|x) = \frac{1}{1 + \exp(Af(x) + B)} \tag{7}$$

where A and B are scalar parameters, estimated using *maximum likelihood method*. Using this approach, we can now 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.

6) 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), the goal of RFE is to select features by recursively considering smaller and smaller sets of features. 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 procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

III. Evaluation

A. 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-python-api framework provided by GDB [14]. We have used Capstone [22], which is a lightweight disassembly framework, for disassembling the crashing instruction and branch records available from LBR. We have also made use of readelf [23] utility to extract the permissions of allocated memory segments. We use a PIN [24] 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 [25] and do not affect the results in any way except for increasing 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 [26] 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 [27], a robust machine learning library available in python, for classification, crash prioritization and feature ranking.

B. Results and Discussion

1) Parameter tuning and model selection: Table II gives *weighted f1-measure* for SVM models with different parameters values for C and γ. Based on the scores, we find that the best classifier consists of *rbf* kernel with parameters $C = 100$ and $\gamma = 0.001$ (highlighted in table).

TABLE II
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)

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 use grid search with cross-validation to determine the optimal parameter values for the given *development* dataset. 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.

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 III. 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 IV, that numerically describes the classifier prediction with actual class of the samples.

TABLE III
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 IV
Confusion Matrix

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

Figure 3 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.7).

Table V 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

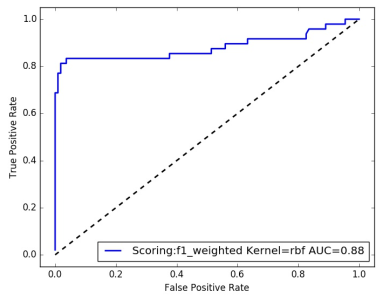


Fig. 3. ROC curve on evaluation set with selected classifier.

TABLE V
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

testcases to be 0.451 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.

3) Crash Prioritization: As discussed above, we can utilize the probability assigned for a test sample as a measure of its priority to fix the bug. Consider the Listing 3 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 11.

```
 1 int main() {
 2   char a[40][40], temp[40];
 3   long int b[40];
 4   int i=0;
 5   while(1) {    // infinite loop
 6     long int j;
 7     for(j=0;j<=strlen(temp);j++) {
 8       if(temp[j]>=65 && temp[j]<=90)
 9       temp[j]+=32;
10     }
11     strcpy(a[i],temp);  /* crash! out of bounds write in
            array a*/
12     scanf("%ld",&b[i]); /* user input in b[i] */
13     i++;
14   }
15   ...
16   return 0;
17 }
```

Listing 3. Prioritizing a non-exploitable crash

This naive example is *non-exploitable* because the input of attacker does not reach the crash site as input in *b* is totally disjoint from array *a*. We also used *DTA* to confirm that crash instruction is not affected by user input. But this case is predicted *Exploitable* by both !exploitable and Exniffer. However, if we rank the crashes based on probability estimates, we find that this case has a probability of 0.573 only, which does not indicate a severely exploitable case as predicted by !exploitable.

In another example from evaluation set, we consider an *exploitable* scenario. In the Listing 4 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 7 since coder forgot to re-initialize *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 the crash at a lower level of severity by classifying it as *Probably_Exploitable*.

```
 1 void update(int **B,int row,int col,int *input,int k){
 2   int i,j;
 3   i=k;
 4   for(i=0;i<row;i++)
 5     printf("\n%d %d",i,input[i]);
 6   for(j=0;j<2;j++)
 7     B[i][j]=input[j];    // crash! coder forgot to re-
          initialize i
 8 }
 9 int main(){
10   int len1=30,len2=3,i,k=0,testcase,l=0;
11   int *input = (int *)malloc(sizeof(int)*3);
12   int **B = (int **)malloc((len1) * sizeof(int *));
13   for (i=0; i<(len1); i++)
14     B[i] = (int *)malloc((len2) * sizeof(int));
15   scanf("%d",&testcase);
16   while(l<testcase){
17     for(i=0;i<3;i++)
18       scanf("%d",&input[i]);
19     if(unique(B,len1,len2,input,k)){ // unique -> true
20       update(B,len1,len2,input,k);  // update called
21       k++;
22     }
23     l++;
24   }
25   ...
26   return 0;
27 }
```

Listing 4. Prioritizing an exploitable crash

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

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. 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

TABLE VI
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 Violation Signal
49.	Loop
17.	#Operands = 3+
46.	Conditional Jump
43.	Floating point exception Signal

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 VI, 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 if there is *immediate* operand in the crashing instruction that gives an indication of in-feasibility 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) 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 VII gives the prediction of *!exploitable* on the entire dataset using a confusion matrix representation as percentage for respective classes. The first column is divided as per the categories assigned by !exploitable. Predicted categories are shown 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

TABLE VII
CONFUSION MATRIX FOR !EXPLOITABLE

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

that *!exploitable* performs better on *non-exploitable* cases as compared to *exploitable* scenarios. 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. 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.

IV. RELATED WORK

Due to a large frequency of bugs and substantial efforts needed to fix them, the problem of crash classification is considered to be very important and there have been several attempts to address this issue. Novel et. al. proposed ReBucket [28], a method for clustering crashes in buckets based on call stack similarity. Bucket information was used to prioritize debugging efforts. Although exploitability was not considered but their system improved existing Microsoft's Windows Error Reporting (WER) [29]. In their paper on predicting top crashes, Kim et. al. [30] use machine learning for the purpose of crash 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.

VDiscover [17] extracts dynamic features from sequences of calls to C standard library and then predicts exploitability using machine learning. Although VDiscover false positive rate is low, it requires long sequences of dynamic features and there is a scope of improvement with respect to true positive rate. ExploitMeter [31] 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 III-B5, !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 indicated ineffectiveness of similar static features.

CREDAL [32] aims to localize memory corruption vulnerabilities by using core-dump along with source code to give more insight to developer. 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. MAYHEM [16] and Lai et al. [33] perform automatic exploit generation with concolic execution. Miller et al. [5] analyze 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, the runtime to generate execution traces (in taint-flow analysis for example), is very large (see Table V) and hence all these systems perform offline crash analysis. Although we did not have automatic generation of exploit as an application in mind while developing Exniffer, we do propose that these approaches can be used for complete analysis of only most severe crashes such as those predicted exploitable by Exniffer.

V. Conclusion

Today there are millions of bugs reported across thousands of software products for example 1.7 million bugs are reported in more than 12,300 projects at Launchpad since 2004 [2]. Bugs resulting in these crashes can be exploitable and have to be fixed in time to avoid security ramifications. 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 LBR that incurs negligible runtime-overhead and is suitable for production systems. The results show that the approach is effective in exploitability prediction and crash prioritization. In future, we plan to expand the set of features by engineering static and dynamic features to improve accuracy further. We will also increase the size of dataset to perform a more exhaustive testing of the system.

Acknowledgement

We would like to thank the anonymous reviewers for their comments. This work was partially supported (for one author) by the Netherlands Organisation for Scientific Research through grant NWO 639.023.309 VICI (Dowsing).

References

[1] J. Barnes, "Gem #30: Safe and secure software: Introduction," Ada Lett., vol. 29, no. 1, pp. 45–47, Mar. 2009. [Online]. Available: http://doi.acm.org/10.1145/1541788.1541799

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

[3] 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. IEEE, 2015, pp. 478 – 483.

[4] Z. Puhan, W. Jianxiong, W. Xin, and Z. Wu, "Program crash analysis based on taint analysis," in Proc. IEEE International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 2014, pp. 492 – 498.

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

[6] J. Arulraj, G. Jin, and S. Lu, "Leveraging the short-term memory of hardware to diagnose production-run software failures," in ASPLOS, NY, USA, 2014, pp. 207–222.

[7] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," in Proc. conference on Virtual Execution Environments, 2012.

[8] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in Proc. OSDI'08, 2008.

[9] M. E. Joorabchi, M. Mirzaaghaei, and A. Mesbah, "Works for me! characterizing non-reproducible bug reports," in Proc. Working Conference on Mining Software Repositories, 2014.

[10] K. Walcott-Justice, J. Mars, and M. Lou Soffa, "Theme: a system for testing by hardware monitoring events," in Proc. of ISSTA'12. NY, USA: ACM, 2012, pp. 12–22.

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

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

[13] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in Proc. S&P'10. IEEE, 2010, pp. 317–331.

[14] "Gdb: The gnu project debugger," https://www.sourceware.org/gdb/.

[15] S. Rawat and L. Mounier, "Finding buffer overflow inducing loops in binary executables," in IEEESERE'12, 2012, pp. 177 – 186.

[16] S. Kil-Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in Proc. S&P'12, 2012, pp. 380 – 394.

[17] 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 CODASPY'16. NY, USA: ACM, 2016, pp. 85–96.

[18] 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. S&P'16, 2016, pp. 110 – 121.

[19] "Online compiler and debugging tool," http://ideone.com/recent.

[20] J. Mercer, "Functions of positive and negative type and their connection with the theory of integral equation," in Philos. Trans. R. Soc. London, 1909, p. 415446.

[21] H. T. Lin, C. J. Lin, and R. C. Weng, "A note on platts probabilistic outputs for support vector machines," in Machine Learning, 2007, pp. 68:267–276.

[22] "Capstone: The ultimate disassembler," http://www.capstone-engine.org/.

[23] "Readelf: Gnu binary utilities," https://linux.die.net/man/1/readelf.

[24] 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. PLDI'05, 2005, pp. 190–200.

[25] 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. SOSP'15. NY, USA: ACM, 2015, pp. 344–360.

[26] "Taint analysis pin tools," https://github.com/JonathanSalwan/PinTools.

[27] "scikit-learn: Machine learning in python," http://scikit-learn.org/stable/.

[28] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Novel, "Rebucket a method for clustering duplicate crash reports based on call stack similarity," in Proc. ICSE'12, 2012.

[29] 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 Proc. SOSP. NY, USA: ACM, 2009, pp. 103–116.

[30] 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, 2011, pp. 430 – 447.

[31] G. Yan, J. Lu, Z. Shu, and Y. Kucuk, "Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability," in Proc. IEEE Symposium on Privacy-Aware Computing, 2017.

[32] 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. New York, NY, USA: ACM, 2016, pp. 529–540.

[33] 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, 2014, pp. 270 – 289.