Toward large-scale vulnerability discovery using Machine Learning

Gustavo Grieco[§]
gg@cifasis-conicet.gov.ar

Guillermo Luis Grinblat[§]
grinblat@cifasis-conicet.gov.ar

Lucas Uzal[§]
uzal@cifasis-conicet.gov.ar

Sanjay Rawat[‡]
tosanjayr@gmail.com

Josselin Feist[†]
josselin.feist@imag.fr

Laurent Mounier[†]
laurent.mounier@imag.fr

ABSTRACT

With sustained growth of software complexity, finding security vulnerabilities in operating systems has become an important necessity. Nowadays, OS are shipped with thousands of binary executables. Unfortunately, methodologies and tools for an OS scale program testing within a limited time budget are still missing.

In this paper we present an approach that uses lightweight static and dynamic features to predict if a test case is likely to contain a software vulnerability using machine learning techniques. To show the effectiveness of our approach, we set up a large experiment to detect easily exploitable memory corruptions using 1039 Debian programs obtained from its bug tracker, collected 138,308 unique execution traces and statically explored 76,083 different subsequences of function calls. We managed to predict with reasonable accuracy which programs contained dangerous memory corruptions.

We also developed and implemented VDiscover, a tool that uses state-of-the-art Machine Learning techniques to predict vulnerabilities in test cases. Such tool will be released as open-source to encourage the research of vulnerability discovery at a large scale, together with VDiscovery, a public dataset that collects raw analyzed data.

1. INTRODUCTION

In spite of the progress made in programming languages and software engineering techniques, most of the programs we routinely use (from operating system components to main office or web applications) still contain numerous bugs. However, some of these bugs are clearly more dangerous than the others: the ones which may affect the security of the whole system, hereafter referred to as software *vulnerabilities*. As a consequence, a serious issue for software editors is not only to find bugs, but also to identify which ones correspond to vulnerabilities and require in-depth analysis to estimate their dangerousness, and if necessary, rapidly distribute some adequate patch.

Nevertheless, vulnerability detection is not a simple operation. As has been pointed out in [1],

> "*The* ***defect*** *caused an* ***infection****, which caused a* ***failure*** *and when we saw the failure we tracked the infection, and finally found and fixed the defect.*"

In the context of vulnerability discovery, a failure (i.e., an observable incorrect program behavior) could be a crash. Tracking the infection is possible by *monitoring* the program execution until it finally reaches the defect, i.e. some code calling to an insecure library function. We can observe that all of the three points are related to each other in a way that the presence of one can be used to infer the presence of the other. In other words, a defect will manifest itself in the infection in a very *peculiar* way, which in turn, will lead to a failure.

Some static analysis techniques have been proved successful in finding classical programming flaws, like buffer overflows or null-pointer dereferences, but they suffer from a high percentage of *false positives*. More importantly, only a few tools are able to operate on the binary code. As a result, one of the most effective vulnerability detection techniques still relies on large fuzzing campaigns, feeding the target program with various inputs in order to produce *crashes* that need to be (manually) analyzed afterwards. This is a time-consuming activity. For instance, an operating system like Debian contains more than 30,000 programs and 80,000 bug reports. Methodologies and tools for an OS scale program testing in a limited time budget are still missing. Therefore, there is a strong need for techniques to be used as *fast predictors*, to quickly identify which programs are more likely to contain a vulnerability, in order to direct the fuzzing process.

Given the complexity of modern software, the relationship between defect, infection and failure is not easy to notice, especially by a human analyst. Machine learning and data mining techniques [2] have been used to learn such subtle relationships (dependencies) in a wide range of applications [3, 4, 5], when the complexity involved is too high. As a result, in this work, we resort to the application of Machine Learning techniques to learn such dependencies in the case of a failure.

The objective of our work is to make a step in this direction by presenting a scalable machine learning approach that uses lightweight static and dynamic features to predict if a test case is *likely* to contain a software vulnerability. As far as we know, this is the first large scale study on vulnerability discovery for binary only programs.

1.1 Contributions

The main contribution of this paper is to demonstrate the feasibility of a large-scale study of binary programs in order to predict vulnerabilities according to some procedure to perform vulnerability discovery. In order to build a predictor, we started defining and evaluating different sets of features that can be automatically extracted from binary programs. Such features are designed to be scalable: they are extracted using very lightweight static and dynamical analysis.

To show the effectiveness of our approach, we set up a very large experiment to detect easily exploitable memory corruptions using 1039 Debian programs obtained from its bug tracker. To perform a reasonable evaluation of our methodology, we collected 138,308 unique execution traces and statically explored 76,083 different subsequences of function calls. We managed to predict which programs contained dangerous memory corruptions with a 55% of accuracy and which programs resulted robust with a 83% of accuracy.

We also developed and implemented VDiscover, a tool that uses state-of-the-art Machine Learning to predict vulnerabilities in test cases. Our tool will be released with an open-source license to encourage the research of vulnerability discovery at large scale, together with VDiscovery, a public dataset that collects raw analyzed data.

The paper is organized as follows. We dedicate Sec. 2 to explain the background on vulnerability discovery. Later, we overview the proposed methodology in Sec. 3 and we explain it in detail in Sec. 4. Data generation and feature extraction is presented in Sec. 5. Then, Sec. 6 is devoted to introduce the Machine Learning techniques used in this paper. Experimental setup is detailed in Sec. 7 and results are presented and discussed in Sec. 8 followed by a survey of related work in Sec. 9. Finally we draw some conclusions and point possible future work directions in Sec. 10.

2. BACKGROUND

Many different vulnerability discovery procedures (VDP) has been proposed in the Computer Security literature to detect potentially vulnerable issues in software. As expected, every VDP has particular requirements and biases to identify (specific) vulnerabilities. In this section, we highlight the attributes of different VDP proposed by several authors.

2.1 Fuzzing and Smart Fuzzing

Currently, one of the most effective approaches to find vulnerabilities in large software is based on fuzzing techniques [6], i.e., feeding the target application with *unexpected inputs* and looking for abnormal program termination. The crucial step in fuzzing is clearly to choose relevant unexpected inputs, i.e., likely to reveal potential vulnerabilities. Several techniques can be used.

One of the simplest techniques is *random mutation* of known correct inputs. It requires only a basic knowledge of the target application. However, most of the mutated inputs are likely to be rejected in the early steps of the program execution either at parsing or because of checksum verification.

To overcome this problem, another input generation technique is to better control the mutations using some knowledge about the input format, like in grammar-based fuzzing [7]. However, this technique is effective only with a high level of expertise on the target application.

Such *black-box* fuzzing techniques are rather easy to implement and they are highly scalable since they do not involve complex computations nor heavy program monitoring techniques. Nevertheless, they suffer from two drawbacks: first, they do not allow to control the program execution and second, huge fuzzing campaigns are required to obtain valuable results. Furthermore, the crashes obtained should be processed *a posteriori*, first to filter redundant information (crashes resulting from the same bugs), and second to sort out between harmless bugs and more serious ones. This operation requires a high-level of expertise and is really time-consuming.

To overcome these limitations, some *white-box* fuzzing approaches have been proposed [8, 9]. The underlying idea is to generate the application inputs with the help of its code.

Clearly, the benefit of these "smart-fuzzing" techniques is to better control the program exploration according to a given objective (e.g., either maximizing code coverage, or focusing on specific parts, more likely to be vulnerable). Hence, many tools have been developed in this direction (Klee [10], TaintScope [11]) and their ability to find vulnerabilities has been illustrated on several case studies.

Moreover, some works make use of concolic execution for vulnerability detection [12, 13, 14].

2.2 Static Analysis

Historically, static analysis tools were used to prove the absence of bugs inside a program [15][16], and they proved to be particularly effective in specific application domains like embedded systems or aeronautic applications. Their main advantage is clearly to provide sound reports on the whole set of program executions. However, using these techniques on more general-purpose software is much harder and time consuming. This is essentially due to the over-approximations performed during the analysis to preserve soundness, since static analysis is undecidable [17]. Providing good approximation techniques for general program constructions (involving dynamical memory allocations and complex data structures) is hard, and therefore the number of false positives produced is usually huge.

However a few works have been proposed to apply static analysis techniques in the context of vulnerability detection. They offer lightweight static analysis [18, 19], either on source or binary code, and they generally focus in a specific kind of vulnerability [20]. Also, they sacrifice soundness in order to serve as "bug finders", allowing to better control the trade-off between precision and scalability.

Nevertheless, even if it appears that pure static analysis techniques are not precise enough to be used as standalone

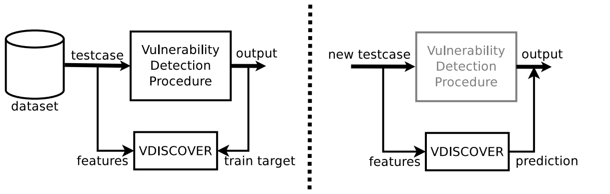

Figure 1: Summary of train and recall phases in VDiscover

general vulnerability detection techniques, they offer highly valuable preprocessing information to combine with other approaches like concolic execution or Machine Learning.

3. OVERVIEW

Our study aims to define and evaluate some machine learning techniques that can be used to predict whether a binary code may contain certain vulnerabilities. We propose a methodology that works in two phases: training and recall.

In the training phase, a large amount of test cases are collected from the binary code in a *training dataset*. These test cases are characterized by static features extracted from the disassembled binaries and dynamical features extracted from its execution analysis. These features are based on the use patterns of the C standard library. Additionally, test cases are evaluated using a vulnerability detection procedure: such procedure flags as vulnerable or not every test case in the train dataset. The objective of this phase is to use the extracted features and the vulnerability discovery procedure to train a predictor using supervised machine learning techniques.

After that, in the recall phase, a *trained classifier* is used to predict if new test cases, extracted from new programs, will be flagged as vulnerable or not. Later, a flagged test case can be prioritized in further analysis. It is also important to note that our approach is not replacing the outcome of a vulnerability detection procedure. Figure 1 summarizes both phases of our approach.

3.1 Building a Predictor

Our tool aims to deal with a very large number of test cases to decide which ones should be further analyzed to look for security vulnerabilities. Needless to say, we want our predictor to distinguish between flagged and unflagged test cases as correct as possible. In this work, accuracy is measured in terms of the errors predicting flagged test cases (false negatives, also called type II error) and the errors predicting unflagged test cases (false positives, also called type I error). VDiscover aims to minimize both types of errors during the training phase. On the one hand, the reduction of false positives allows to discover more vulnerabilities in a shorter time. On the other hand, reducing false negatives decreases the number of misses vulnerabilities in this predictive analysis.

The extraction and processing of our features to predict also included some distinctive design principles. Such principles are:

1. **No source-code required**: Our features are extracted using static and dynamic analysis for binaries programs, allowing our technique to be used in proprietary operating systems.

2. **Automation**: Some Machine Learning applications rely on heavily engineered features to obtain a good performance. This typically requires a human expert to review candidate features before the training phase. In this work, we will focus only on feature sets that can be extracted and selected automatically, given a large enough dataset.

3. **Scalability**: Since we want to focus on scalable techniques, we only use lightweight static and dynamic analysis. Costly operations like instruction per instruction reasoning are avoided by design.

4. METHODOLOGY

In order to show experimental results on the performance of VDiscover to predict vulnerabilities in new test cases, we need a concrete vulnerability detection procedure (VDP) and a dataset to train our tool. In particular, we evaluated our technique using a simple fuzzer to detect easily exploitable memory corruptions on a large corpus of test cases. Despite the evaluation of our technique will be limited to the use of a simple fuzzer as VDP, it is important to note that detection of this kind of vulnerability is not a fixed component of VDiscover. In fact, the adaptive computation performed by machine learning techniques provides a convenient machinery to predict the presence of vulnerabilities using another VDP.

4.1 Detecting Memory Corruptions

Our vulnerability detection procedure comprises two components: a fuzzer to mutate the original test case and a dynamic detection module to identify easily exploitable memory corruptions.

We used a simple fuzzer to explore a large number of variations of a test case. It performs only two types of mutations: replacement of one byte by another and expansion of a random byte at some position in the original input. Using it, we defined a fuzzing campaign that mutates and executes each test case 10,000 times, large enough to catch some interesting memory corruptions.

We also need to define when a program is vulnerable to a potentially or easily exploitable memory corruption and how that can be detected automatically. Detecting this type of vulnerability is not as easy as it sounds, especially without source code or debugging information. We define two ways of detecting memory corruptions, through explicit and implicit evidence of them.

Explicit signs of stack and heap memory corruptions are:

1. Corruption of stack memory: Some of the Debian binaries are compiled with stack protections provided by the GNU C standard library, so in case of stack corruption such protections will abort the execution. Additionally, we can inspect the call stack when a program

crashed, looking for return addresses of called functions. If we find at least one invalid return pointer, then we immediately conclude that the stack frames were corrupted.

2. Corruption of heap memory: We take advantage of the heap consistency check made by the GNU C standard library. If we find a call to abort related with this check, we conclude that a heap corruption happened.

Implicit hints of memory corruptions include:

1. Corrupted or unexpected arguments in some functions: A few key functions like `strcpy`, `memcpy`, `fread`, `fwrite`, among others have its arguments inspected during execution. For example, a call to memcpy indicates a potential memory corruption if it has a very large count parameter value (e.g., bigger than 2^{24}).

2. Corruption of a pointer to a function: If a crash is detected, we inspect if the instruction pointer is pointing to an invalid or a non-executable page.

4.2 Memory corruption for fun and profit

To illustrate how important it is to prevent memory corruption, we present a small example of this issue that can easily be exploited to hijack the control flow of a faulty program. The vulnerable condition in this example can be detected using the procedure detailed in 4.1 and the affected program is flagged as vulnerable in VDiscovery.

We will show in detail the analysis of a crash in xa, a small cross-assembler for the 65xx series of 8-bit processors (used in computers such as the Commodore 64). This command line utility is located in the Debian package xa65. The version 2.3.5 can be crashed using an unexpectedly large input in one of its arguments. The insecure code is shown in Figure 2a. This crash is the result of a buffer overflow caused by the improper use of sprintf at lines 9–10. It is worth mentioning that this memory corruption is *not* directly exploitable by overwriting the return address of a function call since the invocation of sprintf will write in global memory (at lines 2–3).

An alternative way to exploit this bug is available, since a pointer to a *FILE* structure is controllable by an attacker. A large input in the sprintf argument can be used to overflow the array, and rewrite the fperr *FILE* pointer. By abusing the fact that the *FILE* structure contains a virtual function table, we can craft a fake *FILE* structure with a pointer to our own payload. Once this layout is placed in memory, we should just wait for the program to execute a fprintf (line 19) with our malicious *FILE* structure (and to use our fake virtual function table), which happens just after, inside the logout function. This technique is not new at all, in fact, it was well known by Greek hackers more than 10 years ago [21]. Despite that, it still works today when it is tested on a fully updated installation of Debian Sid.

We will also illustrate how VDiscover extracts patterns to detect vulnerable programs using a small piece of x86 assembly code from the xa utility shown in Figure 2b, since this program contains many examples of vulnerable code. Such code reads data from the environment (line 1) and calls to strcpy (lines 5–7) without checking the size of the input variable.

5. DATASET

It is not possible to learn from a single test case using a Machine Learning approach. A large amount of them are needed during a training procedure. Also, additional example cases are required to measure a trained predictor. Unfortunately, at the time of writing, we found no suitable dataset to perform the evaluation of our technique.

The need for these cases were our main motivations to construct VDiscovery, our dataset. It was created by analyzing 1039 test cases taken from the Debian Bug Tracker. Every test case uses a different executable program and they are distributed over 496 packages. They were originally packed along their inputs by the Mayhem team using their symbolic execution tool and submitted to the Debian Bug Tracker [22]. The programs comprised in VDiscovery are quite diverse and included data processing tools from scientific packages, simple games, small desktop programs and even an OCR. Using VDiscover, we can unpack and parse the necessary input sources (command line, standard input, files, etc.) in order to instantly reproduce each test case.

5.1 Classes

After using the previously defined vulnerability discovery procedure described in Sec. 4.1, test cases are divided in two classes: *flagged as vulnerable* and *not flagged as vulnerable*. A program is said to be *flagged as vulnerable* if there is at least one trace that exhibits a vulnerable memory corruption pattern. As expected, our dataset suffers from a severe class imbalance [23]. Only 8% of the total of programs are flagged. This is an issue we have to tackle before the predictor starts learning from it, as explained in Sec. 7.

5.2 Features

In this work, two sets of features are defined and evaluated: dynamic features extracted from the execution of test cases and static features extracted from the binary programs. Both set of features try to abstract the use patterns of the C standard library and they are represented as variable-length sequences. Nevertheless, they aim to capture different aspects of it. On the one hand, static features are extracted by detecting potential subsequences of function calls. On the other hand, dynamic features are captured from execution traces containing concrete function calls augmented with its arguments.

Features are not necessarily correlated with the concrete vulnerability that we are trying to detect. In fact their objective is to provide a redundant and robust similarity measure that a Machine Learning model can employ to predict whether a test case will be flagged as vulnerable or not. Such prediction will be based on previously seen examples during the training phase.

5.2.1 *Static Features*

Static features are supposed to capture information relevant to a whole program, and they should be obtained without

```c
/* global variables */
static char out[MAXLINE];
static FILE *fpout, *fperr, *fplab;
...

int main(int argc,char *argv[])
{
  ...
  sprintf(out, "Couldn't open source \
               file '%s'!\n", ifile);
  logout(out);
  ...
}

void logout(char *s)
{
  fprintf(stderr, "%s",s);
  if(fperr)
    fprintf(fperr,"%s",s);
}
```

(a)

```
1  call getenv
2  test %eax,%eax
3  je @15
4  lea -0x100c(%ebp),%ebx
5  mov %eax,0x4(%esp)
6  mov %ebx,(%esp)
7  call strcpy
8  movl $0x123,0x4(%esp)
9  mov  %ebx,(%esp)
10 call strtok
11 ...
12 ...
13 ...
14 ...
15 ret
```

(b)

Figure 2: Different routines of /usr/bin/xa in C and x86 assembly

running the code on particular inputs. Classical static analysis techniques use graph-based representations to express the code structure, like call graphs, control and data-flow graphs, etc. However, building such structures is costly and not always possible from a stripped binary code.

The approach we propose is to "approximate" a code structure as a set of finite sequence calls to the standard C library. Such a sequence set can be viewed as an abstraction of the program call graph where only some function calls are considered, and where the graph structure is flattened.

These static features can be extracted directly from the binaries using a very lightweight static analysis. First, the binary is disassembled using a linear sweep algorithm. The set I of direct calls to C standard library functions is extracted from the disassembled code. Elements of I will be used as starting points for a random exploration of the program control-flow graph. We build the set S of library function calls by repeatedly using the algorithm described informally below:

1. select an element c of I and insert it in an empty sequence σ;
2. follow the subsequent instructions of c in the disassembled code, and:
   - if a call or jump to a C standard library function is found, append it to σ and continue with the next instruction;
   - if an unconditional jump to the address x is found, continue at address x;
   - if a conditional jump is found, randomly select a branch and continue on this branch;
   - otherwise, skip the instruction and continue with the next one;
3. when a return statement or an indirect jump is reached, the sequence σ is terminated and added to the resulting set S.

As expected, this simple procedure extracts feasible and unfeasible subsequences of C standard library calls by a random walk on a part of the program control flow graph.

Example. In Figure 2b, if we start from the call to getenv at (1), two possible subsequences of C standard library calls can be extracted, according to the conditional jump at (3). The resulting set S is then:

$$\{[\text{getenv}; \text{strcpy}; \text{strtok}; \ldots], [\text{getenv}]\}$$

Computational Cost. The extraction of this kind of features requires to completely disassemble a program: the executable of the analyzed test case. After that, the lightweight static analysis performed is stateless and the random walking only needs to collect a sample of the potential C standard library calls.

5.2.2 Dynamic Features

Dynamic features are supposed to capture a sample of the behavior of a program in terms of its concrete sequential calls to the C standard library. Additionally the final state of the execution is included. Such features are extracted by executing for a limited time a test case and hooking program events, collecting them in a sequence. We define program events as either a call to the C standard library function (abstracted simply as fc_i) with its arguments or the final state of the process:

$$fc_i(arg_1, .., arg_n) \mid FinalState$$

The final state will be analyzed to determine which event will be the last one of a trace. In this work, a program can finish with an exit, an abnormal termination, an induced abnormal termination or, it can run out of time.

$$Exit \mid Crash \mid Abort \mid Timeout$$

An important difference with the static features is the amount of data that can be potentially extracted from a test case. Even for small programs, the collection of traces can create a very large dataset, since a simple loop can be unfolded into an arbitrarily long sequence of events.

Nevertheless, it is really difficult for a Machine Learning classifier to discover useful relations using these traces of raw events. The fact that the arguments of function calls are low level computational values, like pointers and integers, becomes an issue for learning patterns in traces. There are two important reasons for this.

In first place, it will induce an enormous range of different values (e.g., 2^{32} in 32-bit). If we want to train our classifier with discrete sequences of events, it is essential to drastically reduce the range of different events. And in second place, these values convey very little information by themselves. So, it is necessary to augment them with relevant information in order to be able to learn from them.

To address these two issues, we decided to *tag* every argument value with a suitable subtype. The subtyping relation we defined is shown in Figure 3. It is loosely inspired by TIE [24] and PointerScope [25] subtyping systems for reverse engineering.

In the case of the pointers (`Ptr32`), it is very useful to know the region where they are pointing to: for instance, `HPtr32` indicates heap, `SPtr32` stack and `GPtr32` global memory. Also it is relevant to know if they are dangling (`DPtr32`) or null (`NPtr32`).

And in the case of integers (`Num32`), since they convey even less information than pointers, it is useful to know if they are zero, small, large or very large. To formalize this kind of imprecise knowledge, our approach is to divide them in logarithmic buckets so a subtype of the generic integer type gives an idea of how large it is, e.g `Num32B`n indicates a 32-bit number between 2^n and $2^{(n+1)}$. In case of looking for suspiciously small or large arguments, for example, reading or writing bytes, it is useful to use such subtypes.

Example. After executing the vulnerable code in Figure 2b, VDiscover captures the following piece of trace presented here in comparison with the ltrace [26] output:

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

Computational Cost. The extraction of this kind of features requires the execution of a test case. In order to do that, the analyzed binary and its dynamically linked libraries are instrumented to detect calls to C standard functions. Whitelisting is also performed, discarding internal function calls from the libraries contained in the C standard library package. Such restriction in our dataset aims not only to minimize the cost of instrumentation but also to reduce the complexity of the resulting features. It therefore allows different Machine Learning techniques to learn from this kind of features more easily.

It is also worth to mention that we designed VDiscover to require the collection of **a single trace** during the recall phase. In our experiments, such trace is collected using the fuzzer detailed in 4.1 to reduce the bias of the test cases (generated using a symbolic executor). Otherwise, the original

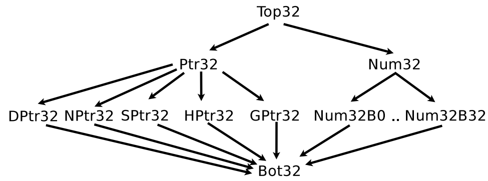

Figure 3: Subtyping relation used to process the function arguments in the traces

test case should be used.

6. A MACHINE LEARNING APPROACH

6.1 Models

A wide variety of Machine Learning and statistical models address the classification problem. We can sort these models ranging from those with few parameters, linear boundary surface and easy to train, to models with many parameters, nonlinear boundary surface and very hard to train. The *logistic regression* model [2] is between the former. It models the log conditional probability of category outputs (given the inputs) as an affine transform of the inputs. In our case these inputs are either the static or the dynamic features after being preprocessed.

The logistic regression model can be extended to a nonlinear model by adding one or more intermediate nonlinear transforms –known as *hidden layers*– between the input and the affine transform. These layers also consist of an affine transform plus an element-wise nonlinearity. The resulting model is the most common version of a *multilayer perceptron* (MLP). Notice that when using an MLP model we must choose certain design parameters (*hyperparameters*) such as the number of layers and the dimension of each intermediate representation (number of *hidden units*).

The parameters (or *weights*) of each layer are obtained by maximizing the log-likelihood over the training data. This formulation casts the learning process as an optimization problem over the weights. The optimization is performed using the *stochastic gradient descent* (SGD) algorithm which is commonly used for training artificial neural networks [2]. The SGD algorithm is suitable for handling large datasets since training examples are seen in small batches. The optimization algorithm has its own hyperparameters that must be chosen beforehand together with model hyperparameters.

Additionally, we complete the list of Machine Learning models considered for comparison with the *random forest* method [27]. Random forest is an ensemble of decision trees trained on bootstrap data sets with a random selection of features. This model is a widely adopted method for classification due to its resistance to overfitting and the small number of hyperparameters that are required to optimize during the training phase.

6.2 Validation and regularization

All Machine Learning methods are susceptible to overfitting, i.e. explaining certain particular features present in a finite training set which damage the performance for new and unseen examples. This behavior implies that an error

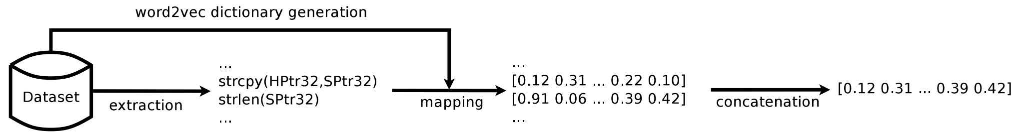

Figure 4: Preprocesing of dynamic features using word2vec

estimation over the training set is overly optimistic. Therefore a separated set of unseen samples is required for an unbiased error estimate. Furthermore if we want to use this estimation for choosing the best set of hyperparameters we must use a *validation* set for this purpose and leave an unseen *test* set for the final unbiased error estimate [2]. This means that we must split the available data in three parts: the training, validation, and test sets.

The validation set is used for monitoring the error over unseen samples during training. By stopping training when validation error reaches a minimum some degree of overfitting can be avoided. This *early stopping* technique [2] also biases the model to having small weights since they are initialized with small random values.

Another way for improving generalization is the recently proposed *dropout* training technique [28]. This technique has been widely adopted in recent years for improving generalization error over a large variety of neural networks [29, 30]. We applied it to both logistic regression and MLP.

7. EXPERIMENTAL SETUP

7.1 Data Preprocessing

Before starting to train the vulnerability predictor of VDiscover, the features of our dataset were preprocessed. Data preprocessing is essential to be able to train and test Machine Learning models out of the box. This procedure should also reduce the dimensionality of the sequential data in VDiscovery, since training Machine Learning models require to use fixed-length inputs.

In order to process the different sets of features, we used two procedures taken from the text processing and mining field. We started considering each trace as a text document. Such approach is very similar to the ones already used to deal with traces in other works [31, 32]. Also, for each preprocessing procedure, different parameters were used, since they can have a large impact in the accuracy of a trained classifier:

- bag-of-words: this widely used [33] preprocessing technique was applied. For each feature set, we used 1-grams, 2-grams and 3-grams to get suitable representations to train and test our vulnerability predictor.
- word2vec: this preprocessing technique was recently designed for learning continuous vector representations [34] of words in large text datasets. We selected it since it was successfully used in a variety of text mining applications [35, 36]. As shown in Figure 4, word2vec was used to generate a vectorial representation of all possible events. Then, for each trace, a vector was formed selecting events from the beginning and the end of each trace. We experimented with 20, 50, 100 or 200 vectorized events concatenated in order to characterize complete executions.

A critical issue in our dataset is the class imbalance. We addressed it using a well tested solution known as *random oversampling* [23] in order to facilitate the learning process of different classifiers.

7.2 Training Procedure and Models

In order to perform a valuable evaluation of the different features and classification methods, we processed several training datasets with only static or only dynamical features, so every set of features was evaluated independently.

For each feature set, a total of 40 predictive experiments were made splitting the dataset in three **completely disjoint program sets**: train, validation and test. As we explained in Sec. 6, such condition is essential to report honest results. We want our trained classifiers to generalize beyond the examples available in the training set of programs.

In our experiments, we trained several machine learning classifiers: logistic regression, MLP of single hidden layer and random forest. Mature and well tested software packages like scikit-learn [37] and pylearn2 [38] were employed to train and test different classifiers.

7.3 Error Evaluation

The use of a highly imbalanced dataset requires additional care when accuracy is computed after the prediction of a test set. Otherwise, a trivial classifier predicting every program as unflagged will report a misleading accuracy. In order to use a sensible test error measure, we account false positives and false negatives as percentages. To obtain a single error percentage, we can average false positives and false negatives into a *balanced test error*. Unless stated otherwise, we refer to this quantity as *test error*.

8. RESULTS AND DISCUSSION

8.1 Results

Tables 1a, 1b and 1c summarize the test errors on vulnerability detection. Our most accurate classifier was a **random forest trained using dynamical features**. To show the accuracy of such classifier, we present the corresponding confusion matrix in Table 3a in terms of the test cases that VDiscover detects as flagged or not. Using the most accurate classifier, we can estimate the reduction in the effort needed to discover new vulnerabilities.

Input	**Logistic Regression**	**MLP**
200 events	38% ± 1	35% ± 1
100 events	**34**% ± 1	37% ± 1
50 events	35% ± 1	36% ± 1
20 events	37% ± 1	35% ± 1

(a) Dynamic features (word2vec)

Input	**Logistic Regression**	**Random Forest**
1-grams	40% ± 1	32% ± 1
1–2-grams	40% ± 1	**31**% ± 1
1–3-grams	40% ± 1	**31**% ± 1

(b) Dynamic features (bag-of-words)

Input	**Logistic Regression**	**Random Forest**
1-grams	**37**% ± 1	43% ± 1
1–2-grams	**37**% ± 1	41% ± 1
1–3-grams	**37**% ± 1	40% ± 1

(c) Static features (bag-of-words)

Table 1: Average test error of vulnerability prediction using VDiscover

Variable	Importance
fflush:0=Ptr32	6%
StackCorruption	6%
MemoryCorruption	4%
malloc:0=Num32B24	4%
fread:1=Num32B8	3%
memset:0=GPtr32	3%
memset:1=Num32B0	2%
strcat:1=SPtr32	2%
strcat:1=GPtr32	2%
exit:0=Num32B32	2%
strncpy:0=SPtr32	2%
strrchr:0=SPtr32	2%

(a) With relevant features

Variable	Importance
strrchr:1=Num32B8	11%
printf:0=GPtr32	9%
_IO_getc:0=Ptr32	4%
malloc:0=Num32B32	3%
getenv:0=GPtr32	3%
strcasecmp:1=GPtr32	3%
open:1=Num32B8	3%
fprintf:0=Ptr32	3%
Timeout	2%
strcasecmp:0=SPtr32	2%
fopen:0=SPtr32	1%
malloc:0=Num32B16	1%

(b) Without relevant features

Table 2: A comparison of variable importance between trained vulnerability predictors with and without features relevant to memory corruptions

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

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

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

8.2 Feature Analysis and Robustness

After performing the series of experiments detailed in this section, the results suggest that the proposed methodology was appropriate for the prediction task. Nevertheless, it is important to investigate further how the trained Machine Learning model is differentiating and characterizing test cases.

In order to evaluate the robustness of the best predictor, it is important to know which features are more important and how they are used to predict. Despite model interpretability is a very desirable property, there is no general approach to understand how and why trained models take decisions in the recall phase. Fortunately, we can easily extract an importance score for each variable in the feature set from a trained random forest [27].

To analyze the model robustness, first we want to define a special subset of dynamic features: the *features relevant to the specific VDP*. In our experiments, these features are defined according to the procedure to detect easily exploitable memory corruption as explained in 4.1. They include features associated with certain function calls (e.g. strcpy, memcpy, etc.) as well as the final state indicating if the there is a crash, abort or exit.

Without loss of generality, we decided to analyze one of the simplest models we trained: a random forest using bag of words (1-gram) of dynamic features that achieved a reasonable accuracy (32%). The most significant variables are shown in Table 2a where *relevant features* are in bold.

As you can see, *relevant features* are widely used as the most important ones for prediction. Still, the resulting classifier is not completely dominated by only a few features. Nevertheless, at this point, it is critical to know if *relevant features* are responsible for all or most of the accuracy in the prediction task. If this is the case, the predicted model is just looking for trivial evidence to detect vulnerabilities in memory corruption.

A simple yet effective way to estimate the real importance of a set of features is to remove them from the original dataset and re-train the predictor. Such procedure will force the model to predict without them. We can estimate how important *relevant features* are in the prediction comparing the accuracy of the re-trained predictor. Interestingly enough, after re-training without the *relevant features*, the test error in prediction is only marginally higher (35%) than our best predictor. Most significant variables for the re-trained predictor are shown in Table 2b.

Using this simple procedure, we show that the resulting predictor is robust, in the sense that the removal of some features still allows to get a reasonable prediction for flagged test cases. We hypothesize that the model is taking advantage of the generality of the features to detect behaviorally similar test cases. Using such similarity allows it to predict correctly instead of looking for features relevant to the memory corruption itself.

8.3 Speed

VDiscover is implemented in Python using the python-ptrace binding [39] and GNU Binutils. It is designed to scale avoiding to use extremely slow operations like instruction per instruction execution. Nevertheless, in terms of code optimization there is very little work done.

The extraction of dynamic features is performed for every analyzed binary hooking its global offset table and detecting calls to C standard library functions. A very lightweight value analysis of the arguments of every call is also performed. The instrumented executions are on average only 7 times slower, a trade-off we considered acceptable given the obtained results.

As it was explained in Sec. 5, static feature extraction is defined as a stateless procedure, in which a part of control flow graph is random-walked to collect subsequences of function calls. Nevertheless, there is no need to explicitly reconstruct the control flow graph, so the feature extraction cost is dominated by the parsing and disassembly of the analyzed binary. Fortunately, we employ GNU Binutils which is highly optimized for this task, taking no more that a few seconds to extract static features in a modern desktop computer.

It is also worth to mention that VDiscover works with ELF x86 binaries on Linux. Despite the fact that the current implementation is limited to that platform, there is no reason to think that the same approach cannot work in other operating systems without ptrace (e.g. Windows) if there is support for breakpoints and peek/poke memory of a process.

8.4 Comparison

As far as we know, no other technique was designed to perform vulnerability discovery at a large scale without source

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

(a) VDiscover

	Flagged	**Not Flagged**
Exploitable	14%	5%
Probably Exploitable	21%	18%
Probably Not Exploitable	43%	59%
Unknown	22%	18%

(b) !Exploitable

Table 3: Comparative between VDiscover and !Exploitable predictions of test cases

code, so we have not found a fair approach to compare our work with others. Nevertheless, we found a suitable tool to give a fast evaluation of the bug severity in memory corruptions: !Exploitable. It also works performing a lightweight analysis of a test case. This tool was originally developed by Microsoft [40] and later adapted to run in Linux using GDB by the CERT [41].

Unlike our tool, !Exploitable *requires* a crash to analyze its final state and the failing assembly instruction. It outputs an *exploitable category* according to heuristics encoded in prefixed rules: *exploitable*, *probably exploitable*, *probably not exploitable* or *unknown*. After running all the test cases in VDiscovery, we collected the categories answered by *!Exploitable*. A summary of our experiments is shown in Table 3b.

In order to make a sensible comparison between !Exploitable and VDiscover, it is necessary to map such categories to binary answers about the exploitability of the programs in our dataset. A reasonable choice is to consider vulnerable programs if they are flagged as *exploitable* or *probably exploitable*. Computing the balanced error from Table 3b results in 44% of test error while VDiscover is at 31%.

On the one hand, our tool represents a substantial improvement in the prediction to discover new vulnerabilities. On the other hand, !Exploitable analyzes crash executing programs at native speed and requires no training at all. Unfortunately, in our experiments, the accuracy of !Exploitable is close to a random guess (e.g. a test error of 50%) and thus results in poor performance to predict vulnerability at a large scale. It is important to note that this comparison is limited to the VDP selected for the experimental evaluation.

8.5 Data Limitations

As expected, lightweight extraction of features from binary programs has several limitations: a prediction error of 31% in the testing of VDiscover shows that there is room for improvement. The confusion matrix from Table 3a presents a visible unbalance between the accuracy of the detection of flagged and non-flagged test cases. We hypothesize that the relatively small number of flagged test cases available during the training phase is limiting the classifiers accuracy.

It seems natural to try to combine both features sets to improve prediction accuracy. Unfortunately, the results of this strategy are quite disappointing. The test prediction error were similar to the ones obtained using only static features. We found no effective way of combining different sets of features to improve prediction accuracy. We believe that the train phase is affected by the fact that static features are less diverse than dynamic ones because they are shared by all the traces of the same program. In other words, the number of independent training samples is reduced to the number of different programs as soon as we include the static features. Machine Learning algorithms assume independence of the training examples and, in the presence of this (artificial) persistence in the static feature values for a large set of flagged test cases, it tends to use this subset of features for discrimination. Therefore, the generalization capabilities are not better than using only static features.

The use of features also has its limitations. For instance, static features cannot be used to analyze different test cases of the same program, since the program is only statically analyzed without taking into consideration its actual input. This limitation did not affect our experiments, since our dataset only contains one test case per program but it is certainly an issue if VDiscover is used to evaluate a large set of test cases. Additionally, static features should be considered naturally more imprecise than dynamic features in general, since every non-trivial binary program contains many distinctive procedures.

The use of dynamic features has its own limitations: learning from traces is difficult because they have variable size and they can contain different amounts of useful information. For example, a complex program can use libraries. As expected, each library will have their own intrinsic patterns and a trace from such program will contain interleaved events from different libraries making pattern recognition a very challenging task.

9. RELATED WORKS

A very close work, albeit for a different problem of malware analysis, is reported in [42]. Similar to our approach, its authors have used static and dynamic features to form vectors of binary features of malware behavior. This vector is used in a supervised Machine Learning algorithm to produce rules for further classification. In spite of the reported similarities, there are differences in the way the vectors are generated. Unlike the reported work, our static and dynamic feature extraction is much lighter and hence introduces a very small overhead. It is important to note that extracting features from the actual malware process and code is a very challenging task, since most of these programs are packed or encrypted, and designed to avoid running normally under a virtualized environment. Therefore, we do not claim that our technique can be easily adapted to analyze malware.

Another close work is reported in [43], where the idea is to detect vulnerable code patterns from vulnerable source code. Similar to our approach, the main idea is to form a vector of characteristics that capture the semantic and syntactic structure of the function code and then use a machine learning approach to classify new functions. However, unlike ours, the proposed technique works with the source code of the program and has the different objective of finding vulnerable code patterns.

It is also worth to mention that there are plenty of approaches reported in the past wherein machine learning techniques are applied for attack detection (in the context of intrusion detection systems) [44, 45, 46]. However, we would like to point out that though the objective of finding *subtle and hidden* dependencies by using Machine Learning remains the same, our work involves a much fine-grained approach to extract feature vectors, which is more tuned towards the problem at hand i.e., classifying the bug on the basis of its severity.

10. CONCLUSIONS AND FUTURE WORK

As we have shown in previous sections, the large scale prediction of programs flagged and unflagged as vulnerable using static/dynamical features is feasible even without source code. The reached error rate of 31% suggests that there are patterns in the features that can be detected using a machine learning algorithm. Given such promising results, we are already working on the evaluation of different VDP as well as some directions that we plan to explore in the near future.

On the one hand, regarding static features, it could be a good idea to search for similarities between program slices, e.g. by creating a tree representing the possible sequences of C standard library calls. Using this tree could help to detect similar behavior during the training of the classifier.

On the other hand, regarding dynamical features, it is expected that interesting patterns could appear at different locations along the traces. Convolutional neural networks (CNN) [2] have been developed to model patterns in images with translation invariance along the image 2D array. This dramatically reduces the number of parameters to train with respect to a standard multilayer perceptron, improving generalization capabilities. We then expect that a 1D version of a CNN can improve the current performances over traces. There is a promising ongoing work in this direction.

In conclusion, this study shows that Machine Learning applications on a large scale binary-only vulnerability detection can have the potential to significantly increase the number of vulnerabilities found at operating system scale.

11. ADDITIONAL AUTHORS

12. REFERENCES

[1] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann Publishers Inc., 2005.

[2] C. M. Bishop *et al.*, *Pattern recognition and machine learning.* springer New York, 2006, vol. 1.

[3] H. Drucker, S. Wu, and V. N. Vapnik, "Support vector machines for spam categorization," *Neural Networks, IEEE Transactions on*, vol. 10, no. 5, 1999.

[4] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, 2006.

[5] A. Genkin, D. D. Lewis, and D. Madigan, "Large-scale bayesian logistic regression for text categorization," *Technometrics*, vol. 49, no. 3, 2007.

[6] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley Professional, 2007.

[7] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *SIGPLAN Not.*, 2008.

[8] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Sage: whitebox fuzzing for security testing." *Commun. ACM*, 2012.

[9] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. IEEE Computer Society, 2009.

[10] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI.* USENIX Association, 2008.

[11] T. Wang, T. Wei, G. Gu, and W. Zou, "Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution." *ACM Trans. Inf. Syst. Secur.*, 2011.

[12] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. IEEE Computer Society, 2012.

[13] 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," *Reliability, IEEE Transactions on*, March 2014.

[14] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Commun. ACM*, 2014.

[15] P. Cousot, R. Cousot, J. Feret, L. Mauborgne *et al.*, "The astreÉ analyzer." ser. Lecture Notes in Computer Science. Springer, 2005.

[16] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto *et al.*, "Frama-c - a software analysis perspective." ser. Lecture Notes in Computer Science. Springer, 2012.

[17] W. Landi, "Undecidability of static analysis." *LOPLAS*, 1992.

[18] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis." *IEEE Software*, 2002.

[19] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. IEEE Computer Society, 2014.

[20] S. Rawat and L. Mounier, "Finding buffer overflow inducing loops in binary executables," in *Proceedings of Sixth International Conference on Software Security and Reliability (SERE).* IEEE, 2012.

[21] killah@hack.gr, "File Stream Pointer Overflows Paper," http://www.ouah.org/fsp-overflows.txt, 2003.

[22] M. Team, "Reporting 1.2K crashes," https://lists.

debian.org/debian-devel/2013/06/msg00720.html, 2013.

[23] H. He and E. A. Garcia, "Learning from imbalanced data," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 21, no. 9, 2009.

[24] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs."

[25] M. Zhang, A. Prakash, X. Li, Z. Liang, and H. Yin, "Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis," 2012.

[26] J. Céspedes, "ltrace," http://www.ltrace.org, 2014.

[27] L. Breiman, "Random forests," *Machine learning*, 2001.

[28] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," 2012.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012.

[30] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*. IEEE, 2014.

[31] H. Pirzadeh, A. Hamou-Lhadj, and M. Shah, "Exploiting text mining techniques in the analysis of execution traces," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, Sept 2011.

[32] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[33] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., 2005.

[34] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[35] L. Wolf, Y. Hanani, K. Bar, and N. Dershowitz, "Joint word2vec networks for bilingual semantic representations," *International Journal of Computational Linguistics and Applications*, vol. 5, no. 1, 2014.

[36] S. P. F. G. H. Moen and T. S. S. Ananiadou, "Distributional semantics resources for biomedical text processing."

[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, 2011.

[38] I. J. Goodfellow, D. Warde-Farley, P. Lamblin, V. Dumoulin *et al.*, "Pylearn2: a machine learning research library," 2013.

[39] V. Stinner, "python-ptrace," http://python-ptrace.readthedocs.org, 2014.

[40] Microsoft Security Engineering Center (MSEC) Security Science Team, "!Exploitable," http://msecdbg.codeplex.com, 2013.

[41] Jonathan Foote, "CERT Triage Tools," http://www.cert.org/vulnerability-analysis/tools/triage.cfm, 2013.

[42] I. Santos, J. Devesa, F. Brezo, J. Nieves, and P. Bringas, "Opem: A static-dynamic approach for machine-learning-based malware detection," in *International Joint Conference CISIS'12-ICEUTEt'12-SOCOt'12 Special Sessions*, ser. Advances in Intelligent Systems and Computing. Springer Berlin Heidelberg, 2013, vol. 189.

[43] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *Proceedings of the 5th USENIX Conference on Offensive Technologies*, ser. WOOT'11. USENIX Association, 2011.

[44] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, ser. SP '96. IEEE Computer Society, 1996.

[45] S. Rawat, V. P. Gulati, and A. K. Pujari, "Transactions on rough sets iv." Springer-Verlag, 2005, ch. A Fast Host-based Intrusion Detection System Using Rough Set Theory.

[46] T. G. and C. P., "Learning rules from system calls arguments and sequences for anomaly detection," in *Proc. ICDM Workshop on Data Mining for Computer Security (DMSEC)*. Springer, 2003.