Diploma Thesis

Automated Extraction of API Usage Patterns from Source Code for Vulnerability Identification

written by
Fabian Yamaguchi
Matrikel: 306784

Supervisors: Prof. Dr. Klaus-Robert Müller, TU Berlin
Dr. Konrad Rieck, TU Berlin
Felix 'FX' Lindner, Recurity Labs GmbH

Technische Universität Berlin, Fachgebiet Maschinelles Lernen
Institut für Softwaretechnik und Theoretische Informatik
January 2011

Abstract

In the past decades, numerous methods have been proposed to partially automate the discovery of those well understood classes of software security vulnerabilities, which lend themselves easily to generalization. Despite these efforts, vulnerability identification remains an inherently creative process, in which an auditor must carefully examine the intentions of the developer and the code chosen to express them, and it is in the subtle differences between intentions and implementations that vulnerabilities are commonly found. Modularization, the definition of auxiliary functions and the presence of conventions imposed by external interfaces, naturally lead to the formulation of these intentions by means of application-specific, often sparsely documented programming patterns, which in consequence give rise to application-specific sets of vulnerabilities.

In this work, the use of unsupervised learning algorithms as developed in the field of machine learning is proposed to exploit the statistical properties of the target code-base and arrive at a vectorial representation of source-code based on dominant programming patterns suitable to assist in manual source-code auditing. Furthermore, by measuring distances within the corresponding vector space, functions within a code-base sharing similar programming patterns can be efficiently located, thereby allowing for *vulnerability extrapolation*, i.e. the discovery of zero-day vulnerabilities similar to known vulnerabilities in terms of application-specific programming patterns. The proposed method's ability to group sets of related functions is quantitatively evaluated on a data set assembled from popular open-source code-bases and its practical use is demonstrated by example. In particular, a zero-day vulnerability in the popular media decoding library FFmpeg found by extrapolation of a known vulnerability is presented and its exploitability for arbitrary code execution on a recent Ubuntu Linux installation is demonstrated.

Zusammenfassung

In den vergangenen Jahrzehnten wurden eine Vielzahl von Methoden entwickelt um die Entdeckung jener sicherheitsrelevanter Schwachstellen in Software Systemen partiell zu automatisieren, die sich einer einfachen Generalisierung zugänglich machen. Trotz dieser Bemühungen bleibt die Identifikation von Schwachstellen ein inhärent kreativer Prozess, in dem Auditoren sorgfältig die Unterschiede zwischen den Absichten des Entwicklers und dem Code, der diese umsetzen soll, untersuchen müssen. Modularisierung, die Definiton von Hilfsfunktionen und das Diktat von Konventionen durch externe Schnittstellen, führen zu einer Formulierung dieser Absichten in applikationsspezifischen, oft nur spärlich dokumentierter Programmiermuster, die applikationsspezifische Gruppen von Schwachstellen bedingen.

In dieser Arbeit wird vorgeschlagen unüberwachte Lernalgorithmen aus dem Forschungsgebiet des Maschinellen Lernens zu verwenden, um die statistischen Eigenschaften von Code-Basen dahingehend auszunutzen, dass eine vektorielle Darstellung des Quellcodes basierend auf dominanten Programmiermustern entsteht, die geeignet ist eine manuelle Untersuchung von Code in Hinblick auf Schwachstellen zu unterstützen. Außerdem wird demonstriert wie durch Messung von Abständen im entsprechenden Vektorraum Funktionen identifiziert werden können, die gemeinsame Programmiermuster enthalten, sodass *Schwachstellen Extrapolation* ermöglicht wird, d.h. die Entdeckung von Zero-Day Schwachstellen, die einer bekannten Schwachstelle in Hinblick auf die API Nutzung ähnlich sind. Eine quantitative Evaluierung der vorgeschlagenen Methode in Hinblick auf ihre Verwendbarkeit zur Gruppierung von verwandten Funktionen wird auf einem Datensatz durchgeführt, der aus Code bekannter Open Source Projekte zusammengesetzt wurde. Weiterhin wird die praktische Anwendbarkeit der Methode anhand eines Beispiels demonstriert. Insbesondere wird in diesem Zusammenhang eine Zero-Day Schwachstelle in der bekannten Mediendekoder Bibliothek FFmpeg präsentiert, die unter Verwendung des Verfahrens durch Extrapolation einer bekannten Schwachstelle gefunden wurde. Weiterhin wird die Ausnutzbarkeit der Schwachstelle zur beliebigen Code Ausführung auf einem zeitgemäßen Ubuntu Linux System bewiesen.

Contents

Abbreviations

API	Application Programming Interface
ASLR	Address Space Layout Randomization
CPU	Central Processing Unit
DNS	Domain Name System
CSV	Comma Separated Values
GUI	Graphical User Interface
IDF	Inverse Document Frequency
PCA	Principal Component Analysis
SQL	Structured Query Language
SVD	Singular Value Decomposition
TF	Term Frequency
TF-IDF	Term Frequency-Inverse Document Frequency
XSS	Cross Site Scripting

Notation

$\mathcal{A}$	An alphabet
C	Covariance Matrix
D	Dimensionality of data set
d	Dimensionality of approximation
e_a	A unit vector
$\mathcal{F}$	A feature space
$\mathcal{I}$	An input space
k	A kernel function
L	A Lagrangian
$\mathcal{L}$	A language
N	Number of data points
X	A data set containing vectorial elements
x_n	The n'th data point, a vector
$\bar{x}$	The center of mass of X
Φ	A non-linear mapping
σ_a	Variance of data along e_a
$< x, y >$	Scalar product between x and y
$\frac{\partial f}{\partial x}$	Partial derivative
$*$	The Kleene-Star
$\#_{w(x)}$	Number of occurrences of w in x

Erklärung

Die selbstständige und eigenhändige Ausfertigung versichert an Eides statt

Berlin, den July 26, 2011

_____

Fabian Yamaguchi

Acknowledgment

As I complete this work, my time as a student at TU Berlin ends and I would like to take this as an opportunity to thank a number of people who have accompanied and in many cases guided me throughout this journey. First and foremost, I would like to thank my parents, two people I can truly look up to. You've shown me that education is but a format of something much more valuable and that it would be downright stupid to miss out on it just because you dislike the color of the paper it is packed in.

I would also like to thank my supervisor Dr. Konrad Rieck for supporting this work in all stages, for sharing his experience and for a number of extremely helpful discussions. This would have all been a lot more difficult without you. Thank you FX and the team at Recurity Labs for giving me the best student-job ever. The work at Recurity has definitely been the B-Sides of my education.

Thank you, darp, for the great time we had studying together and for silently exchanging my last beer and the one after that for glasses of water. You've definitely saved me from some headaches. And finally, thank you Jana for being much more important to me than any of the work I do could ever be.

CHAPTER 1

Introduction

1.1 Motivation

With the introduction of data processing systems into our daily lives, we have become dependent on the correct operation of software. However, software is usually imperfect and contains a variety of different defects, some of which can be exploited by attackers to disrupt operation, access sensitive information or take over the data processing system. Clearly, from a defensive point of view, there is an interest in identifying and resolving these *software vulnerabilities* before they are exploited.

In general, identifying software defects as well as proving their exploitability is non-trivial and manual auditing of the target software system is to date considered to be one of the most effective strategies for vulnerability discovery. [AHLR07] Unfortunately, given the sheer amount of code and the complex structure of many software systems, manual auditing is a time consuming and error prone task. Hence, there is a demand for tools, which assist in the process of code understanding in vulnerability identification.

In this work, we cast the problem of vulnerability discovery into one similar to problems commonly encountered in the natural sciences where vast amounts of data are preprocessed using statistical methods from the field of machine learning to simplify subsequent analysis by domain experts.

Algorithms from the field of machine learning have been successfully applied in the context of IT security before, in particular for malware detection and classification [NKS05, KM06, BCH+09, BOA+07, RHW+08] as well as intrusion detection [Rie09, KV03, WS04, II07]. In the past years, a large body of publications both dealing with the design of such systems

and the possibilities to circumvent them [PDL+06], has been produced. However, to our knowledge, the application of machine learning algorithms to problems of offensive security such as vulnerability identification has to date only been scarcely explored. A notable exception is the field of automatic protocol reverse engineering, which has applied their results to craft input for fuzz testing [Cui07, WCKK08].

The method presented in this thesis demonstrates a successful application of unsupervised machine learning to vulnerability identification. In particular, we demonstrate how *vulnerability extrapolation*, i.e. the discovery of zero-day vulnerabilities based on known vulnerabilities, can be performed using our method.

1.2 Contribution

In this work, a vectorial representation of source code *based on dominant API usage patterns* is presented. We arrive at this representation by considering the problem of finding dominant API usage patterns to be similar to that of finding topics of discussion from sets of text documents, which has been extensively studied in the field of natural language processing. In this domain, Principal Component Analysis is commonly applied to project documents and words into a vector space such that the projections of words often used in combination as well as the documents containing these words are close to one another. In summary, our contributions are the following:

1. We demonstrate how source code can be mapped to a vector space in practice by generating suitable robust parsers from island grammars, which extract function boundaries and API symbols from all functions present in a code base.
2. We then illustrate the effects of Principal Component Analysis as applied to the generated data sets and show that semantic similarities between functions and API symbols can be measured in the resulting lower dimensional sub-space. Furthermore, we provide empirical evidence for our method's ability to group sets of semantically related functions by performing a quantitative evaluation on a data set composed of real production code.
3. An application of our method to vulnerability extrapolation is then demonstrated and a zero-day vulnerability discovered using this method is presented.
4. We close by showing that the generated vectorial representation of source code is suitable for source code browsing in general and can assist in the analysis of API usage patterns, a necessary step to perform *taint analysis* (see section 2.2.2) on application-specific API usage patterns.

1.3 Related Work

Detecting patterns in source code to assist in the improvement of software quality has been of interest in the past. In particular, the detection of exact or slightly modified copies of code fragments, so called *code clones*, has been extensively studied in an attempt to reduce unnecessary duplication of code and thereby lower required maintenance effort. In this context, it has also been observed that code clones often contain the same software defects, which led to the direct application of code clone detection to the discovery of software defects based on known defects, a practice closely related to the idea of vulnerability extrapolation discussed in our work. Furthermore, it has been observed that software defects commonly occur as a result of violating implicitly defined application-specific programming rules, similar in concept to API usage patterns as exploited by our method.

In this section, existing research in programming rule extraction and code clone detection is presented and differences and similarities to our work are discussed.

Data Mining For Programming Rules Engler et al. were among the first to link a significant amount of bugs in large software systems to the violation of implicitly introduced and largely undocumented system-specific programming rules[ECH+01]. Following prior work [ECC00] on the development of system-specific compiler extensions used to detect violations of programming rules, Engler proposed a method, which allowed a user-supplied rule template to be automatically tailored to specific systems. Templates suitable for the description of common system programming errors such as null pointer dereferences and failures to lock shared resources were presented and the method's applicability to real system code was demonstrated by example. A notable shortcoming of this approach is its inherent limitation to such programming rules, which can be easily described using general rule templates. Therefore, the auditor performs the necessary generalization and must provide the method with a narrowed view of the applications bug classes before the analysis, which stands in contrast to the exploratory nature of vulnerability discovery. The method's success thus depends highly on the auditors ability to predict bug classes likely to exist in the target system.

In contrast, Li and Zhou present a tool called *PR-Miner* based on frequent itemset mining[LZ05], which allows for automated extraction of programming rules and detection of violations without the need for rule templates. The authors propose to use programming elements such as functions, variables and data types, which are frequently used in combination, as the basis for programming rule extraction. The most notable drawback of this approach is the underlying optimistic assumption that if programming practices are followed often enough, they constitute a valid programming rule. As such, a set of functions following the same invalid programming practice will lead to the inference of a programming rule, which by definition none of the members of the set violate.

Williams et al.[WHM05] as well as Livshits and Zimmermann[Liv05] propose to mine

software revision histories for commonly made changes. The detection of programming rules not related to bug fixes therefore becomes less likely. On the downside, only programming rules violated in the past can be detected, making the discovery of previously unknown bug patterns infeasible.

Code Clone Detection. The detection of code fragments created by *copy&pasting* of other code fragments has been an ongoing research topic for decades. A thorough evaluation of existing methods has been provided by Bellon et al. [BKS+07].

Kontogiannis et al.[KDM+96] use metric features such as the number of functions called, the ratio of input to output variables as well as McCabe Cyclomatic Complexity[McC76] to compare functions and detect code clones. Detection is thus loosely based on structural properties of functions rather than symbol usage. A similar but more fine grained approach is taken by Baxter et al.[BYM+98] who base their analysis on abstract syntax trees.

Li et al. present *CP-Miner*, a tool for code clone detection based on frequent itemset mining [LLMZ04], similar to their work on programming pattern extraction. They demonstrate the superiority of their approach to the highly cited tool *CCFinder* presented by Kamiya et al.[KKI02], a token based method, which in contrast to CP-Miner does not take into account the statistics of the code base. Maletic and Marcus present work on the detection of high level concept clones from source code in [MM01], which from an algorithmic point of view is closely related to our work. However, their representation of source code stands in direct contrast to ours. While in their work, detection is based on the comments and identifiers contained in a function and API symbols are discarded, in our work comments and identifiers are discarded while API symbols form the cornerstone of our approach.

This highlights the differences between code clone detection and the discovery of functions sharing programming patterns. While the assumption that comments and names of identifiers have been conserved for *copy&paste* clones may hold, this is not true for functions, which merely share usage patterns.

When applied in the security context, we believe that both *data mining approaches to programming rule detection* and *code clone detection* suffer from significant shortcomings, which motivate further research.

In particular, data mining approaches to programming rule detection are not well suited for vulnerability discovery for the following reasons.

1. No intermediate representation of functions or code fragments suitable for code browsing or the application of machine learning algorithms is generated. Furthermore, it is unclear how similarity between functions or correlations between usage patterns can be measured based on the programming rules extracted in data mining.

2. In production code, the assumption that commonly identified usage patterns are indeed correct usage patterns does not hold. A programmer may make the same mistake over and over again simply because he is not aware of the subtle details of the chosen formulation. We therefore believe that a tool should refrain from deciding which rules are correct and which are not. Instead, a good presentation of the extracted programming patterns should be created, which assists an auditor in this decision.

3. Many programming patterns exist, which when violated do not have implications for the security of the system. For this reason, methods, which attempt to extract and present all programming patterns to an auditor may not be well suited to assist in vulnerability identification. Instead, we believe that methods used in this context must be able to provide immediate answers to user designed queries to allow the auditor to express interest in specific patterns over others.

Code Clone Detection differs from the discovery of API usage patterns in the following ways.

1. While the occurrence of matching names of identifiers, comments and other attributes of function appearance may be good indicators for *copy&paste* operations, they are in general not related to the API usage of a function. Therefore representations of source code created for code clone detection are usually not suited for API usage pattern discovery.

2. While code clones will most likely contain similar API symbols, in the discovery of API usage patterns these repetitions are not the most interesting cases. In fact, finding occurrences of the same usage pattern in different contexts is of greater interest in the code understanding process. As such, the assumption that similar functions have been created by copy&paste operations must not hold. In vulnerability identification, one is not primarily interested in functions implementing similar functionality but in functions implementing functionality by similar means.

1.4 Overview

In chapter 2, we give an overview of the field of vulnerability discovery by describing the most common classes of software security vulnerabilities and outlining methods developed to partialy automate this process. Chapter 3, then illustrates the unsupervised machine learning algorithm *Kernel Principle Component Analysis* and shows how it can be applied to textual data. In chapter 4, the importance of API usage patterns to software security is outlined and the proposed vectorial representation of source code obtained by applying Kernel Principal Component Analysis is presented. We then describe how this representation can be applied to assist in source code analysis. In chapter 5, the application of the proposed method is first demonstrated on a toy problem to develop an intuition for the projection performed by our

method. We then proceed to evaluate the method's ability to group sets of related functions on a real data set. In chapter 6, the process of finding a zero-day vulnerability by *vulnerability extrapolation* is demonstrated by example and the discovered vulnerability is exploited to proof its relevance. We then demonstrate how the method can assist general source code browsing. In chapter 7, we summarize our work, outline its current limitations and suggest future work.

CHAPTER 2

Vulnerability Discovery

A software *vulnerability* is a weakness in a software system, which can be leveraged by attackers to their advantage. Vulnerabilities are often closely related to software *bugs*, subtleties in the implementation leading to unintended, unexpected and usually undesired behavior such as program crashes. [DMS06]

It is however important to note that the majority of bugs do not result in vulnerabilities even if they result in program crashes given that the *denial of service* state itself does not constitute a vulnerability. Particularly as *generic anti-exploitation* technology is deployed in major operating systems, proofing that a program crash can indeed result in a vulnerability has become a non-trivial and time consuming process [Sot09]. Therefore, while automatic detection of program crashes and violations of invariants may be possible, the detection of vulnerabilities has remained a manual process.

Nonetheless, a large number of publications reporting the automated detection of a high number of *potentially exploitable* vulnerabilities has been produced, few of which proof that the supplied bugs are indeed vulnerabilities. Given the fact that the detection of vulnerabilities in general is not possible as follows from Rice's Theorem, these publications usually deal with the detection of specific *vulnerability classes*, which allow for generalization.

In this chapter, we give an introduction into the field of vulnerability discovery by presenting some of the most well known vulnerability classes and methods proposed to detect such vulnerabilities. Furthermore, we outline the limitations of these approaches.

2.1 Vulnerability Characterization

As the broad definition of vulnerabilities given in the previous section suggests, vulnerabilities are diverse in nature and often hard to classify. Nonetheless, terms have been coined to characterize certain classes of vulnerabilities, which occur frequently, in an attempt to simplify discussion.

A brief overview of the most well known classes is given in this section.

2.1.1 Memory Corruption

Most programs must reserve areas of memory to store data provided by the user. If the program fails to limit access to the designated area of memory, a user may be able to *corrupt memory* used to store local or global variables, heap structures or possibly even return addresses of function calls. In many cases, vulnerabilities of this kind can be leveraged such that the program executes attacker-supplied code.

Buffer Overflow Vulnerabilities. Buffer overflows are the most well known sub class of memory corruption vulnerabilities. In this scenario, a program fails to validate whether user supplied data fits into a memory region (i.e. a *buffer*), allowing a user to overwrite memory locations adjacent to the buffer. Buffer Overflows have come to be particularly popular given that generic methods of exploitation applicable for buffers located on the program stack [One96] and heap [ano01] have been developed. However, given recent advances in anti exploitation technology such as *Address Space Layout Randomization* [SPP+04], non executable data regions and hardened heap implementations, these generic methods have been rendered inapplicable on some platforms.

Integer Related Vulnerabilities. As arithmetic is performed with integers of limited range, overflows and underflows can occur. Furthermore, as integers are converted to integers of lower width, truncation can occur. While this does not directly lead to memory corruption, it may trigger other bugs such as heap-based buffer overflows specifically if the integer is related to the number of bytes allocated for a buffer or the number of bytes to copy into a buffer. Methods for the discovery of integer related bugs have been presented in [WWLZ09] and [BCJ+07]. Details on exploiting scenarios commonly encountered in conjunction with integer related bugs can be found in [ble02].

Format String Vulnerabilities. Format String vulnerabilities are another well known sub class of memory corruption vulnerabilities. In this scenario, a user controls a format string passed to utility functions such as *printf*, *err* or *syslog* and it has been shown in [Scu01] that control over the format string can be leveraged by an attacker to write arbitrary data to arbitrary locations in memory. Some operating system vendors have responded by limiting the set of conversion specifiers available in format strings, thus limiting the applicability of known exploitation techniques.

In general, memory corruption vulnerabilities are transformed into write operations to arbitrary locations in memory to allow stored function pointers to be overwritten and thus redirect the execution flow to an attacker-supplied location in memory.

2.1.2 Code Injection

User-supplied data is often used to create database queries expressed in SQL, commands executed on the shell or content to be embedded in a website. In all of these cases, the program must verify whether the user-supplied data contains control characters specific to the high level language used, which an attacker could use to construct commands, which deviate from intended use.

Shell Injection Vulnerabilities. If a user can supply the argument for a program executed on a shell, the developer must either escape or remove control characters such as semicolons and quotes from the argument. If this is not achieved, an attacker may be able to terminate the shell command prematurely and specify subsequent commands to be executed on the shell.

Cross Site Scripting Vulnerabilities. Cross site scripting vulnerabilities also referred to as *XSS* are usually found in web-applications. In this scenario, a user is able to embed script code into a website such that the script seems to stem from the target website. This allows an attacker to bypass the same origin policy imposed by the browser, which protects content from one domain from scripts hosted by another domain.

SQL Injection Vulnerabilities. When user-supplied data is used in the construction of database queries as often seen in web applications, developers need to assure that control characters specific to SQL such as quotes are properly escaped. If this is not achieved, an attacker may be able to modify the SQL query such that arbitrary SQL queries can be executed. As a result, an attacker may be able to retrieve or modify sensitive information.

Directory Traversal Vulnerabilities. Applications, which allow users to retrieve files from a directory of a host system based on user-supplied input, must take care not to allow sequences such as *../* to be specified. Otherwise an attacker may be able to download files outside of the designated directory.

Null Byte Injection Vulnerabilities. Due to the special role the Null Byte plays as a string termination symbol for libc's string operations, special care must be taken when strings are read from binary data streams. In particular when allocation is performed based on the string length, heap based buffer overflows may result.

2.1.3 Design Flaws

Design flaws are different from the categories of vulnerabilities presented so far in that they are extremely dependent on the target application. A design flaw is also commonly referred to as a logical bug and sometimes as a *business logic flaw*. This kind of vulnerability does not stem from vulnerable programming practices but from the design of the application as a whole. For example, consider a DNS server, which does not properly randomize source ports and transaction numbers, such that an attacker can easily forge DNS responses. While the code in this case does exactly what the author has intended, the author did not acknowledge the consequences. Design flaws cannot easily be generalized. Therefore, automating the discovery is in general not possible.

2.2 Methods for Vulnerability Discovery

As Rice's Theorem states, there is no general and effective method to decide whether an algorithm computes a partial function, which has a given non-trivial property. Therefore, an automatic detection of all vulnerabilities of a software system is in principle not possible. Nonetheless, if a class of vulnerabilities can be described in the appropriate amount of detail, partial automation can be achieved. Nonetheless, it has been acknowledged that none of these tools can replace a thorough manual analysis and that manual analysis by domain experts remains one of the most effective methods of vulnerability discovery. Therefore, in our work, we do not try to automate the process of vulnerability discovery and leave much room for the creativity of the auditor, however, we see the potential to assist in vulnerability discovery using statistical analysis. In this section, however, we briefly discuss methods presented in the past to partially automate vulnerability discovery, in particular to highlight limitations of the state of the art and motivate assistive technologies for manual analysis.

A distinction is made between *static* and *dynamic* methods for vulnerability discovery. In essence, methods of static analysis examine the target program's source or machine code for vulnerabilities without executing it, while dynamic methods monitor the programs execution to detect vulnerabilities at runtime.

2.2.1 Fuzz Testing

Fuzz testing, also commonly referred to as *Fuzzing*, is a dynamic method for finding software defects, which when triggered, produce program crashes or other easily monitored effects. The basic idea of fuzzing is to automatically craft input based on an auditor supplied data model, which can be provided explicitly(*generation fuzzing*) or is induced from a data set(*mutation fuzzing*). The aim is to construct data that deviates slightly from valid input such that the target program does not immediately discard the input but may crash as the irregularity is processed.

In the literature, a distinction between black box fuzzing and white box fuzzing is commonly made. While in black box fuzzing, the target system is usually only monitored for program crashes, white box fuzzing techniques monitor which program paths input triggers and adjusts input such that a high amount of code coverage is achieved. In this context, genetic algorithms have been applied [SECZ07, DEP07] to guide the data generation process dynamically.

Fuzz Testing is well suited to assist in the discovery of low hanging fruit vulnerabilities in protocol and file format parsers and has been shown to be especially effective at finding memory corruption vulnerabilities. Also, it is notable that source code is usually not required. However, a notable drawback of most fuzz testing tools is that vulnerabilities, which do not result in program crashes cannot be found. Furthermore, while the process of finding program crashes can be achieved without gaining an in-depth understanding of the target program, verifying whether the crash is indeed associated with a vulnerability cannot.

2.2.2 Taint Analysis

Subsets of well understood vulnerability classes can often be described by the flow of user-supplied data from a set of potential source functions to a set of sink functions, a model used in taint analysis [LL05, WWLZ09]

The idea of taint analysis is to monitor the flow of user-supplied data within the application to identify, which information can be directly or indirectly controlled by an attacker. Vulnerability discovery is implemented based on this approach by supplying a set of data sources and sinks and declaring that if data stemming from one of the data sources propagates to one of the sinks without undergoing validation, this is a vulnerability. Thus undiscovered vulnerabilities are deduced in an automated fashion from a general description of the vulnerability class given in terms of *API usage patterns*.

It is however important to understand, that this approach requires an auditor who can provide a clear description of the vulnerability class as it surfaces in the specific program of interest formulated in the language and using the idioms of the particular program. An auditor must therefore gain an understanding of the API usage patterns specific to the application.

Static [JKK06] and dynamic [NS05] versions of taint analysis have been implemented. Also, the method has been applied to source code of various languages, machine code as well as intermediate representations of machine code generated explicitly for program analysis as mentioned in [BSD10].

2.2.3 Manual Code Review

In a manual code review, an auditor attempts to gain an in-depth understanding of the target program's internals by inspecting source or binary code. While the actions performed in a

manual analysis are application dependent and vary between auditors, common strategies have been outlined in the literature.

In [KLA+04], a distinction is made between *top-down* and *bottom-up* approaches to manual analysis, while it is acknowledged that in most cases a hybrid of the two approaches has shown to be most effective. In a top-down approach, auditors first attempt to find vulnerabilities of specific classes, which can be identified with a limited understanding of the application to then gradually increase the depth of exploration. In contrast, in a bottom up approach, large portions of code are first read to obtain an overview of the program structure. The level of detail is then increased throughout the process.

In [Kle10], security researcher T. Klein describes his favored approach to manual analysis as followed: First all entry points for user-supplied data are determined. Second handling of this data is analyzed line by line to track potential oversights. This methodology is also mentioned as a code comprehension strategy by Dowd et. al. in [DMS06]. The authors further suggest a strategy, which is closely related to the notion of vulnerable API usage patterns as outlined in this work. This is illustrated in the following quote:

> In fact, two of your authors typically start reviewing a new codebase by finding the equivalent of the util/ directory and reading the framework and glue code line by line. This technique has great side benefits for future logic and design review efforts because you pick up the language and idioms of the program and its creators.

The method presented in this thesis assists in the discovery of these *languages and idioms* by automatically extracting common combinations of API symbols.

Manual Code Review, while being time consuming and error prone, is necessary to discover both design flaws and application specific vulnerability classes. Furthermore, to assess whether a bug is indeed a vulnerability, an auditor will have to gain an in-depth understanding of the target application even if an initial bug has been found by automated procedures. It is therefore not clear whether time is well spent developing a sufficiently good data model for fuzz testing or adequately describing a common bug class to apply taint analysis.

CHAPTER 3

Unsupervised Machine Learning

The field of machine learning deals with the development and analysis of algorithms suitable to extract and exploit patterns in data. Machine learning algorithms have been successfully applied to problems of a variety of different fields including object recognition in computer vision, DNA sequence classification in biology and text classification in natural language processing.

A distinction is generally made between unsupervised machine learning algorithms, which operate on unlabeled data sets and supervised machine learning algorithms, which operate on labeled data sets. An unlabeled data set is a set of vectors $X_u = \{x_n\}_{n=1...N}$, $x_n \in \mathcal{F} \subset \mathbb{R}^N$ e.g. a set of measurement vectors taken from a physical experiment while a labeled data set is a set of tuples $X_l = \{(x_n, y_n)\}_{n=1...N}$, $x_n \in \mathcal{F}, y_n \in \mathbb{R}$ e.g. a set of vectors each of which belongs to a given class or is associated with a given value.

Supervised machine learning algorithms are commonly used to induce classifiers, i.e. to create programs, which have *learned* to classify input vectors from a set of examples. In contrast, unsupervised machine learning algorithms deal with the extraction of latent information from unlabeled data sets by making use of the statistical structure of the data. Topics in unsupervised learning include *clustering* where groups of data points are determined, *blind source separation* where mixtures of signals are analyzed to uncover the original source signals and *dimensionality reduction* where a lower dimensional representation of a data set is generated, which preserves relationships between data points in the high dimensional representation as well as possible. For an in-depth discussion of fundamental concepts and algorithms of machine learning, please refer to [DHS01, Bis07] among others.

3.1 Principal Component Analysis

Principal Component Analysis [Hot33], also known as Hotelling- or Karhunen-Loéve Transform is a statistical method applied in a variety of fields for dimensionality reduction, feature extraction and data visualization. Applications include face recognition [TP02], image compression and the identification of topics of discussion from sets of text documents in natural language processing. In the latter case, PCA is applied to a set of text documents to analyze the co-occurrence information of words and project vectorial representations of words and documents into a lower dimensional subspace such that documents containing similar word combinations as well as words used as synonyms are projected onto close-by vectors. In this work, we adapt this approach to identify API usage patterns from source code. Therefore, a basic understanding of PCA is crucial to understand the proposed method. This section briefly outlines PCA. For an in-depth discussion, please refer to [Bs06] or [STC04] among others.

In the literature, two different formulations for the problem addressed by PCA exist, both of which lead to the problem of determining the eigenvectors and corresponding eigenvalues of the covariance matrix.

Consider the following setting: An unlabeled data set of N vectors is given by $\{x_n\}_{n=1...N}$ where $x_n \in \mathcal{I}$ and $\mathcal{I}$ is referred to as an input space. Aligning vectors in columns yields the data matrix $X \in \mathbb{R}^{DxN}$ where D is the dimensionality of the input space.

Maximum Variance Formulation. In the *maximum variance* formulation, PCA seeks particularly descriptive directions in $\mathcal{I}$. To achieve this, directions of highest variance, the so called principle components are extracted. This formulation suits the notion of *feature selection* well: In an experiment where vectorial data has been acquired by measuring different features of the observed system, it is commonly of interest which combinations of features are relevant to describe the system and in contrast, which features merely constitute noise. The leading principal components extracted by PCA are linear combinations of existing features (*complex features*) better suited to describe the data set than the original features. Therefore, a feature selection has been performed based on variance of the data set.

Minimum Error Formulation. In contrast, the *minimum error* formulation highlights PCA's ability to reduce the dimensionality of the data set. In this formulation, PCA seeks a lower dimensional approximation of X, which is optimal in the least mean squares sense. To construct the corresponding optimization problem, consider that each data point x_n can be expressed as a linear combination of D basis vectors $U = \{u_i\}_{i=1...D}$, or formally:

$$x_n = \sum_{i=1}^{D} a_{ni} u_i$$

where $a_{ni} = x_n^T u_i$, i.e. a_{ni} is the projection of the data point x_n onto the basis vector u_i.

Each x_n can then be approximated by

$$\hat{x_n} = \sum_{i=1}^{d} z_{ni} u_i + \sum_{i=d+1}^{D} b_i u_i$$

PCA now seeks a set of basis vectors $\{u_i\}_{i=1...D}$ and coefficients z_{ni} and b_i such that the mean-square error

$$\sum_{n=1}^{N} ||x_n - \hat{x_n}||^2$$

is minimized. Each approximation $\hat{x_n}$ can then be interpreted as the sum of the signal term $\sum_{i=1}^{d} z_{ni} u_i$, which is the optimal d-dimensional approximation of x_n and the noise term $\sum_{i=d+1}^{D} b_i u_i$.

As noted, both formulations lead to the problem of determining the eigenvectors and corresponding eigenvalues of the covariance matrix. We now outline how the maximum variance formulation leads to this problem.

Consider that the data set has been centered by removing the sample mean from each data point, such that

$$\bar{x} = \frac{1}{N} \sum_{n=1}^{N} x_n = 0$$

. The variance along an arbitrary direction in feature space defined by the unit vector e_a is then given by:

$$\begin{aligned} \sigma_a^2 &= \frac{1}{N} \sum_{n=1}^{N} (e_a^T x_n - e_a^T \bar{x})^2 \\ &= \frac{1}{N} \sum_{n=1}^{N} (e_a^T x_n)^2 \\ &= e_a^T \left(\frac{1}{N} \sum_{n=1}^{N} (x_n)(x_n)^T \right) e_a \\ &= e_a^T C e_a \end{aligned}$$

.

One now seeks the unit vector e_a that points into the direction of maximum variance. Formulation as a constrained optimization problem yields:

$$\begin{aligned} \underset{e_a}{\text{maximize}} \quad & \sigma_a^2 \\ \text{subject to} \quad & e_a^T e_a = 1 \end{aligned}$$

Using the method of Lagrange Multipliers, this problem can be recast into an unconstrained form. The Lagrangian is given by

$$L = e_a^T C e_a + 2\lambda_a(1 - e_a^T e_a)$$

.

Setting the partial derivative with respect to e_a to zero, then yields

$$\frac{\partial L}{\partial e_a} = 2Ce_a - 2\lambda_a e_a = 0 \Leftrightarrow Ce_a = \lambda_a e_a$$

, i.e. the e_a must be an eigenvector of the covariance matrix C.

Left-Multiplication of e_a and the application of constraint $e_a^T e_a = 1$ then yields:

$$\underbrace{e_a^T C e_a}_{\sigma_a^2} = \lambda_a \underbrace{e_a^T e_a}_{=1}$$

The variance σ^2 is therefore maximized when λ_a is maximized, i.e. the first principle component is given by the eigenvector e_a of the covariance matrix C with the highest associated eigenvalue λ_a. Additional principle components are then defined to be eigenvectors of the covariance matrix orthogonal to all previous principle components to arrive at a new an orthonormal basis.

3.2 Singular Value Decomposition

Instead of calculating the eigenvectors of the covariance matrix $C = XX^T$, principal components can also be obtained by performing a Singular Value Decomposition of the data matrix X, a relationship we illustrate in this section. This approach has been implemented in the library *SVDLIBC* [Roh], which has been used in this work.

Definition 1. *The Singular Value Decomposition of a real matrix X is given by*

$$X = USV^*$$

where U is an MxM unitary matrix, S is a diagonal MxN matrix containing the singular values of X, which are non-negative real entries $\sigma_i = \sqrt{\lambda}$ satisfying $\sigma_1 \geq \sigma_2 \geq \sigma_3$. V^* denotes the conjugate transpose of V, a unitary NxN matrix.

Definition 2. *A unitary matrix U satisfies $UU^* = U^*U = I_n$ where I_n is the identity matrix in n dimensions, i.e. the inverse of a unitary matrix is given by its conjugate transpose.*

By use of the above definitions, the covariance matrix can be rewritten as

$$C = \frac{1}{N}\left(X^T X\right) = \frac{1}{N}\left(VS^*U^*USV^*\right) = \frac{1}{N}\left(VSS^*V^*\right) = V\Sigma V^*.$$

where $\Sigma = \frac{1}{N}(SS^*)$. Therefore, the eigendecomposition of the covariance matrix given by $C = V\Sigma V^*$ has been indirectly computed by determining the Singular Value Decomposition. To implement dimensionality reduction, only the first k singular values are calculated iteratively and U and V^* are truncated accordingly. The corresponding decomposition is then referred to as a *Truncated Singular Value Decomposition*.

3.3 Kernel Principal Component Analysis

Principal Component Analysis as described in the previous section is subject to several limitations. First of all, PCA is a strictly linear method, i.e. if data lies along a descriptive non-linear direction in feature space, PCA will not be able to represent the data adequately. Second, PCA can only be applied to vectorial data.

Kernel Principal Component Analysis [SSM98] reformulates PCA as a *kernel method* and thereby generalizes PCA to allow for efficient non-linear dimensionality reduction and as required in this work, the application of PCA to structured data such as strings or graphs.

Kernel Methods in general adhere to the following description [STC04].

- Elements of the input space $\mathcal{I}$ are embedded into a vector space $\mathcal{F}$ referred to as *feature space* using a mapping ϕ, which is implicitly defined by definition of a kernel function $k(x_1, x_2) = \langle \phi(x_1), \phi(x_2) \rangle$.
- Linear methods, which have been subject to detailed studies in the past, are now used in feature space, however, by reformulating methods such that they operate exclusively on scalar products $k(x_1, x_2)$, no explicit mapping of vectors from input space to feature space is required. In consequence, the run-time of the algorithm does not depend on the dimensionality of feature space but on the number of data points and PCA can be performed even if ϕ cannot be explicitly defined. Note, however, that while the method is linear, in general the runtime of the algorithm is not.
- Another useful aspect of the reformulation of an algorithm such that it relies only on scalar products is that data items from the input space need not be vectorial as long as it is possible to define a meaningful kernel for elements of the data set.

In this work, only explicitly defined mappings Φ are used. Therefore, Principal Component Analysis can also be implemented by first mapping the data to feature space and then calculating a singular value decomposition of the data matrix as illustrated in the previous section. Note, however, that formulating PCA as kernel method provides a framework, which allows for modification of the proposed method to incorporate information about the sequence of API symbols or the structure of the basic block graph by replacement of the kernel function. Therefore, it provides a more general view of the proposed method.

3.4 String Kernels

In this section, kernels suitable for strings are presented[SRR07]. In our work, source code is first converted into strings, string kernels are then applied to map source code into a feature space and PCA is applied in this feature space.

A generic string kernel is defined by an alphabet $\mathcal{A}$ and a language $\mathcal{L} \subset \mathcal{A}*$ where $*$ denotes the Kleene star. In other words, the language consists of all possible concatenations of elements of the alphabet. The generic string kernel is then given by

$$\begin{aligned} k(x_1, x_2) &= \langle \Phi(x_1), \Phi(x_2) \rangle \\ &= \sum_{w \in \mathcal{L}} \#_w(x_1) \cdot \#_w(x_2) \cdot c(x, w) \end{aligned}$$

where ϕ is an explicitly defined feature map, which maps strings to a high-dimensional feature space

$$\Phi : \mathcal{A}* \mapsto \mathbb{R}^{|\mathcal{L}|},\ \Phi(x) = \left(\#_w(x) \cdot \sqrt{c(x, w)} \right)_{w \in \mathcal{L}}$$

and $\#_w(x)$ is the number of occurrences of the word w in string x while $c(x, w)$ allows for multiplicative weighing. $\#_w(x)$ is also commonly chosen to be a binary function indicating whether w is contained in x at least once.

By choosing different sets for L, different properties of strings can be expressed in the representation. In particular, the following kernels are commonly defined:

Bag-of-Words Kernel. In the bag-of-words representation, L is chosen to be the set of words occurring in any of the strings. Strings are thus mapped to frequency distributions of the terms contained in the overall set of strings. The bag-of-words kernel is thus proportional to the number of words occurring in both strings. Note that the order of occurrences does not affect string representation.

N-Gram Kernel. An n-gram is a sub-sequence of n items, and thus for an n-gram representation L is chosen to be the set of all sub-sequences of length n contained in any of the strings.

All-Sub-String Kernel. In this representation, L is given by the set of all sub-strings contained in any of the strings. To reduce the time needed to generate this representation, it is also possible to choose a maximum length n of sub-strings to consider.

Weighted Degree Kernel. The weighted degree kernel is similar to the n-gram kernel, however, the position of n-grams is also taken into account. L is thus defined to be the set of all n-grams occurring at all possible positions.

Note that kernels are not limited to strings but can also be defined for trees or graphs and furthermore, that it is possible to combine kernels to construct representations, which combine different properties of the data.

3.5 TF-IDF Weighing

Term Frequency - Inverse Document Frequency (TFIDF) [SB88] is a weighing scheme commonly applied to term document matrices in natural language processing. In this scheme, $c(x, w)$ is given by the product of the *term frequency* function TF and the *inverse document frequency* function IDF:

$$\sqrt{c(x,w)} = TFIDF(x,w) = TF(x,w) \cdot IDF(x)$$

where the term frequency is defined to be the number of occurrences of the word w in x normalized on the number of terms in x, i.e.

$$TF(w,x) = \frac{\#_w(x)}{\sum\limits_{w_j \in \mathcal{L}} \#_{w_j}(x)}$$

The inverse document frequency is given by

$$IDF(w) = log\left(\frac{|X|}{|\{x_j \in \mathcal{X} : I(x_j, w)\}|}\right), \; I(x,w) = \begin{cases} 1 & \text{if } \#_w(x) > 1 \\ 0 & \text{otherwise} \end{cases}$$

i.e. the logarithm of the number of documents divided by the number of documents containing the word w.

CHAPTER 4

A Vectorial Representation of Source Code

It is by defining a suitable mapping from code to a vector space that the statistical methods from the field of machine learning become accessible to code analysis, and it is by careful choice of the properties of this mapping that one is able to construct a representation of code, which on the one hand highlights features deemed to be important to the problem while on the other hand achieving invariance towards modification of features suspected to be irrelevant. In this chapter, the rational behind the choice of API symbols as descriptive features of a function in the security context is discussed. We argue that a representation of dominant API usage patterns not only allows occurrences of known vulnerability classes to be efficiently located but also assists in the discovery of new application-specific vulnerability classes. By understanding the discovery of API usage patterns as a problem similar to that of finding topics of discussion in text documents, we then propose an algorithm based on Principal Component Analysis (section 3.1) to arrive at the desired vectorial representation.

4.1 API Symbols and Usage Patterns

The term *Application Programming Interface* (API) is most frequently used to refer to source level interfaces to software libraries, which provide services to a large number of possibly unrelated and independent applications. For that reason the definition of a programming interface plays a significant role in library design.

In a broader sense an API is a source level interface to arbitrary fragments of semantically related code. As noted in [Blo06], since programs are usually modular and inter modular boundaries define APIs, all programs of considerable size necessarily contain their own internal APIs. APIs in this sense may thus exist deeply tangled with the single application that accesses

them. In this setting, the API may have grown unintentionally from a set of utility functions, which have been introduced by the programmer for convenience and to reduce code duplication, leading to a situation where the programmer does not even recognize the API as such. The semantics of APIs of this kind cannot be expected to be well documented or follow clear definitions, an aspect interesting to an attacker.

For the scope of this document, we define the terms *API symbol* and *API Usage Pattern* in the following way:

Definition 3. *Any identifier used by a function within the code base to access functionality provided by an API is referred to as an API Symbol. In particular, names of types and names of functions used in a function call are API Symbols. This includes type-casts.*

Definition 4. *An API Usage Pattern is a set of API Symbols occurring in several functions of a code base. A dominant API Usage Pattern is an API Usage Pattern, which occurs frequently in the given code base.*

4.2 Relevance of API Usage Patterns to Security

As suggested in [Bra07], an Application Programming Interface can be understood as a boundary of competence. It is where the domain knowledge of one expert ends and that of another begins and software developers are commonly advised not to look past the boundary and trust what the interface documentation promises. In contrast, Bratus observes, that it is part of the *security researcher's methodology* to distrust the documentation, to look past the boundary and discover what the code really does as opposed to what it is believed to do. Subtle differences of this kind are commonly the cause for vulnerabilities, as the following classical examples from the C Standard Library *libc* illustrate.

strncpy may not null-terminate strings. strncpy is a string processing routine from libc, which expects three arguments: a pointer to a destination, a pointer to a source and the maximum number of bytes to copy from source to destination. Its purpose is to copy strings, which are represented as null-terminated sequences of non-null bytes throughout libc. While a developer may expect the destination string to always be null terminated given that otherwise, it is no longer a string, this may not be the case. In fact, if the source string is longer than the maximum number of bytes to copy, the destination string will not be null terminated. Exploitable conditions arising from this subtlety have been discussed in [twi00]. The Common Weakness Enumeration has assigned the identifier CWE-170 to vulnerabilities related to this issue. See [Enu] for a thorough discussion and examples.

Requesting zero-sized allocations may not return a NULL pointer. It is commonly overlooked that when requesting the allocation of zero bytes using *malloc*, *realloc* or *calloc*, these routines may return pointers to valid zero-sized heap blocks. In consequence, one

cannot assume that at least a single byte of memory has been allocated if the routines do not return NULL. This subtlety often leads to heap corruption vulnerabilities as presented in chapter 6. Zero-sized allocations have been thoroughly studied in [Van10].

These examples demonstrate the importance of detailed studies of APIs for software security. Furthermore, they show that classes of vulnerabilities can often be directly linked to distinct API symbols. This correspondence has been recognized by practitioners as the development of popular code auditing tools such as *flawfinder* [Whe], *RATS* [FOR] or *ITS4* [VBKM02] suggests. These tools offer databases of API symbols commonly found in conjunction with vulnerabilities and allow a target code base to be scanned for occurrences. Furthermore, the Common Weakness Enumeration uses the umbrella term *API Abuse* (CWE-227) to describe a variety of vulnerabilities, all of which occur due to unsafe API usage patterns.

In the academic world, the connection between API symbols and vulnerability classes has also been recognized and forms the basis for taint analysis. In taint analysis, an auditor describes a class of vulnerabilities by a *source-sink* system defined exclusively by API symbols. If data tainted by an attacker and stemming from one of the sources propagates to a sink without undergoing validation, a vulnerability is detected. The success of this approach has been demonstrated for popular vulnerability classes such as SQL Injection vulnerabilities as they surface using the Java Class Libraries [LL05], Cross Site Scripting vulnerabilities in PHP [JKK06] and integer related vulnerabilities in association with heap management routines from the C Standard Library [WWLZ09]. However, it is important to understand that the success of taint analysis depends on the ability of an auditor to correctly model the vulnerability class by use of API symbols. API usage pattern discovery can therefore be understood as a method to aid in the construction of vulnerability templates for taint analysis.

Finally, practical work by Dowd et al. in [DSD09], which focused on finding vulnerable API usage patterns of the Microsoft Abstract Template Library and the Firefox NPAPI, resulted in the discovery of a class of vulnerabilities specific to a certain set of applications. In [DMS06], Dowd et al. further suggest a code auditing strategy based on studying internal APIs of a program first and then searching for incorrect use of these APIs to find vulnerabilities.

4.3 Proposed Representation

We propose to consider the problem of API usage pattern discovery to be similar to that of finding topics of discussion from a set of text documents, a problem, which has been extensively studied in the field of natural language processing. Classically, this problem has been addressed by applying Principal Component Analysis to vectorial representations of text documents obtained by applying a bag-of-words feature mapping. This algorithm has been referred to as as Latent Semantic Analysis or Latent Semantic Indexing [DDF$^+$90]. Its name arises from the notion of uncovering a latent semantic sub space of reduced dimensionality from the document space.

In detail, the proposed representation is constructed in the following way.

1. We seek a representation of functions solely dependent on the API symbols used. To achieve this, function boundaries are first extracted from all source files to yield the set of strings $\mathcal{S} = \{s_1, s_2, ..., s_n\}$. We then define the language $\mathcal{L}$ to be the set of all API symbols found in the application or alternatively to be the set of all sequences of API symbols up to a specified length. Each function is then represented as an $|\mathcal{L}|$-dimensional vector in a space where each dimension is associated with one API symbol or sequence of API symbols $w \in \mathcal{L}$. Furthermore, the TF-IDF weighing scheme is used.

   Formally, we apply the feature map

   $$\Phi : \mathcal{S} \mapsto \mathbb{R}^{|\mathcal{L}|},\ \Phi(s) = (I(s,w) \cdot TFIDF(s,w))_{s \in \mathcal{S}}$$

   where

   $$I(s,w) = \begin{cases} 1 & \text{if w is contained in s} \\ 0 & \text{otherwise} \end{cases}$$

   and

   $$0 \leq TFIDF(s,w) = \frac{log|\mathcal{S}| - log\big|\{s_j \in \mathcal{S} : I(s_j,w)\}\big|}{\sum\limits_{w_j \in \mathcal{L}} I(s,w_j)} \leq 1$$

   to each $s \in \mathcal{S}$ to obtain the set of function vectors $\mathcal{F} \subset \mathbb{R}^{|\mathcal{L}|}$, a high dimensional unlabeled data set. Observe that by applying TF-IDF weighing, coefficients of all vectors have been normalized such that they vary between 0 and 1.

2. By aligning function vectors in columns, we obtain a sparse symbol-function matrix

   $$X = \big(\Phi(s)^T\big)^T_{s \in \mathcal{S}} = \begin{array}{c} \\ w_1 \\ w_2 \\ \vdots \\ w_{|\mathcal{L}|} \end{array} \begin{array}{c} \begin{array}{cccc} f_1 & f_2 & \dots & f_{|\mathcal{F}|} \end{array} \\ \begin{pmatrix} \Phi(s_1,w_1) & \Phi(s_2,w_1) & \dots & \Phi(s_{|\mathcal{F}|},w_1) \\ \Phi(s_1,w_2) & \Phi(s_2,w_2) & \dots & \Phi(s_{|\mathcal{F}|},w_2) \\ \vdots & \vdots & \ddots & \vdots \\ \Phi(s_1,w_{|\mathcal{L}|}) & \Phi(s_2,w_{|\mathcal{L}|}) & \dots & \Phi(s_{|\mathcal{F}|},w_{|\mathcal{L}|}) \end{pmatrix} \end{array}$$

   .

   Note that the symbol-function matrix not only allows each function to be understood as linear combination of API symbols but also each API symbol as a linear combination of functions yielding the set of API symbol vectors $\mathcal{T} \subset \mathbb{R}^{|\mathcal{F}|}$.

3. Principal Component Analysis is then performed to exploit the statistical properties of the code base and denoise the data set. This step is implemented by performing a truncated Singular Value Decomposition of X using the Lanczos algorithm, an iterative

procedure suited for high dimensional sparse data sets. The decomposition is given by

$$X \approx U\Sigma V^T = \begin{pmatrix} \leftarrow \vec{u_1} \rightarrow \\ \leftarrow \vec{u_2} \rightarrow \\ \vdots \\ \leftarrow \vec{u_{||\mathcal{L}||}} \rightarrow \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_d \end{pmatrix} \begin{pmatrix} \leftarrow \vec{v_1} \rightarrow \\ \leftarrow \vec{v_2} \rightarrow \\ \vdots \\ \leftarrow \vec{v_{|\mathcal{F}|}} \rightarrow \end{pmatrix}^T$$

This decomposition offers a wealth of information about the code base. The columns of the unitary matrix U are the eigenvectors of the covariance matrix $C = X^T X$, i.e. the principal components, which correspond to the dominant combinations of API symbols as present in the code base. These are the basis vectors, which span the principal subspace. Rows of U and the unitary matrix V contain the low dimensional representations of API symbols and functions within the principal subspace respectively while the diagonal matrix Σ contains the singular values of X, which correspond to the square roots of the eigenvalues of the covariance matrix. Recall that the eigenvalues of the covariance matrix correspond to the variances of the principle components and thus allow the importance of combinations of API symbols for the representation of a code base to be measured.

In practice, the information obtained by calculation of the decomposition can be used in the following ways.

1. Functions can be compared to other functions by comparison of the corresponding rows of V using a suitable metric. This forms the basis for vulnerability extrapolation.
2. Functions and API Symbols can be compared by comparison of rows of V with rows of U. This allows the underlying APIs of functions to be exposed. It also enables an auditor to find functions, which best match a set of API symbols.
3. The most dominant API patterns can be extracted by applying a threshold to the basis vectors given by the columns of U.

4.4 Program Architecture

As part of this work, a source code browsing tool was developed to assess the practical benefit of the proposed representation. In this section, the overall architecture of the tool is outlined and the practical considerations, which lead to the chosen implementation are discussed. The tool is composed of two programs. The *Importer* creates the intermediate representation from the target's source code and stores it on hard disk while the *Locator* allows a user to query the information stored by the Importer in a user-friendly manner.

Importer. The Importer parses the source files of the target code base to extract function boundaries and API symbols used in each of the functions. For each source file parsed,

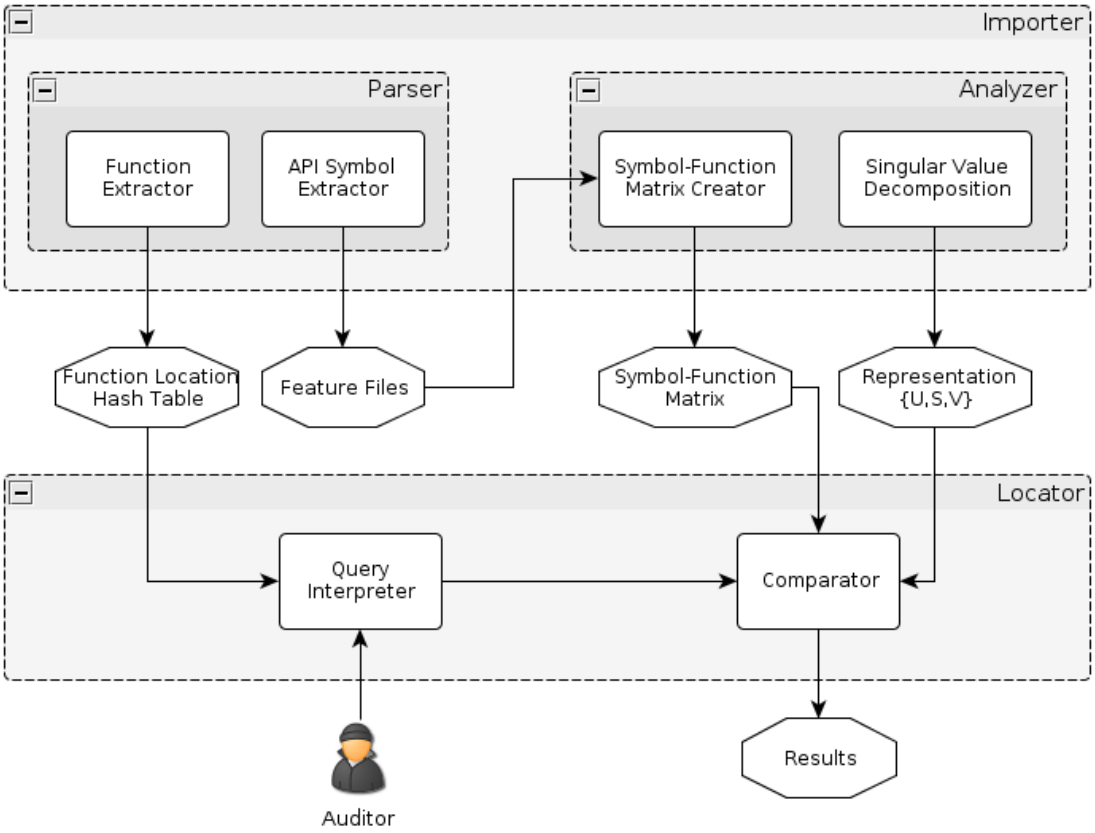


Figure 4.1: Overview of the Architecture of the tool implemented.

a *feature file* is generated. Furthermore, a hash table is constructed, which allows for efficient lookup of the locations of functions by name. Parsers have been defined using grammar definition files for the *ANTLR* parser generator. Refer to section 4.5 for details on this design decision.

Feature Files. Feature files contain the names and line numbers of all identified functions and all API symbols in a *comma separated values(CSV)* format. The acquisition of data from source files, has thus been decoupled from further processing to allow for evaluation of different algorithms on the data sets.

Function Location Hash table. The generated hash table stores lists of function locations by function names. Function locations are encoded in a format inspired by Uniform Resource Locators:

/filename/functionname:lineNumber

This encoding makes two assumptions, which in general may not hold but which are usually not violated in practice: First, each function name must be unique within the scope of a source file. Second, function names may not contain slashes or colons.

Analyzer. The Analyzer constructs a symbol-function matrix from feature files and performs TF-IDF weighing as outlined in the previous chapter. It then performs a truncated

singular value decomposition and stores the symbol-function matrix as well as the matrices resulting from the decomposition on hard disk. To implement this functionality, the Divisi framework [Lab] was used, which in turn makes use of SVDLIBC [Roh].

Locator. The Locator provides a user interface and makes use of the stored matrices to service user requests. A user request is given by names of functions and symbols. The Locator constructs a query vector by adding the vectors associated with the provided names and symbols and determines distances to all other vectors. Functions and symbols associated with vectors that are similar to the query vector are then returned to the user. Functions are optionally written to a file to allow for inspection using a text editor of choice. Similarity is measured using either dot products or the cosine between vectors given by

$$cos(\theta) = \frac{\vec{d_1} \cdot \vec{d_2}}{\|\vec{d_1}\| \cdot \|\vec{d_2}\|}$$

4.5 Fuzzy Source Code Parsing

To implement the proposed mapping, source code must be parsed to identify function boundaries, function-calls and types. One way to approach this problem is by use of a compiler frontend such as Sparse [LT]. [1] However, this choice is associated with a notable drawback: Given the fact that one of the primary tasks of a compiler frontend is to identify whether a program conforms to a language specification or not, in general only complete programs can be parsed using this approach. For example, for the programming language C++, it is not possible to know the semantics of all symbols when header files are missing without the use of heuristics [KL99] . An auditor therefore needs to setup a working build environment, a task trivial in theory but often time consuming in practice. An alternative approach is the implementation of a fuzzy parser [Kop97]. In contrast to parsers based on full-fledged grammars as used in compiler frontends, fuzzy parsers only perform syntax analysis on selected portions of the input and only with respect to certain attributes of interest. The main advantage of fuzzy parsers is that they can operate on incomplete or even incorrect code. However, the price payed is that the generated results cannot be assured to be correct for all programs fulfilling the language specification, given that only limited knowledge about the language is encoded in the fuzzy parser. As such, the fuzzy parser is better understood as a sensor, which extracts possibly noisy measurements from the source code as opposed to a parser in the classical sense.

Moonen [Moo01] offers a formalism for the description of fuzzy parsers known as *island grammars*, which he defines as follows:

An island grammar is a grammar that consists of detailed productions describing certain

[1] Note that GCC-XML [Kin] is not suited for this task because it does not extract function bodies.

constructs of interest (the islands) and liberal productions that catch the remained (water).

Island grammars allow a compact description of fuzzy parsers by means of formal grammars, which can be used as input to parser generators such as GNU Bison [Fou], Elkhound [McP] or ANTLRv3 [Par]. In this work, we have chosen to develop our parser using ANTLR, which allows for a clean transition from the proposed formalism to a grammar input file by use of a language feature known as semantic predicates. [PPQ95]

CHAPTER 5

Evaluation

5.1 A Toy Problem

We conduct a first experiment on a toy problem to help us develop an intuition for the representation of the data and the projection performed by the proposed method. In particular, we demonstrate that given three distinct APIs, the method is capable of projecting both functions and API symbols such that functions sharing an API as well as API symbols constituting the API are close to one another. To achieve this, we define three pairs of functions, each of which makes use of a different API. We then map each function to a vector and apply Principal Component Analysis. Finally, vectors corresponding to the lower-dimensional representations of functions and API symbols are displayed in a scatter-plot to visualize results.

The toy problem consists of three families of functions each containing two members yielding a total of six functions, which contain entirely fictional code constructs. The network-family of functions (*netFunc1* and *netFunc2*) makes use of a network API similar in style to the BSD socket API, the GUI-family (*guiFunc1* and *guiFunc2*) model graphical user-interface code using an API similar in style to the interface exposed by the popular GUI-library GTK while the list-family (*listFunc1* and *listFunc2*) models elements of a custom linked-list implementation.

5.1.1 Setup

The three functions *netFunc1*, *guiFunc1* and *listFunc1* shown in section 5.1.3 are created, which make use of the set of API symbols S_{Net1}, S_{GUI1} and S_{List1} respectively. To create

a problem with three distinct APIs, we choose to impose the constraint that the sets be disjoint, i.e. each API Symbol occurs in exactly one of the functions. This constraint is then slightly loosened by introducing the shared symbol *int*, which is added to each of the sets.

We then proceed to create copies *netFunc2*, *guiFunc2* and *listFunc2* of the three functions and make slight modifications to each copy by adding and removing symbols yielding the sets S_{Net2}, S_{GUI2} and S_{List2} respectively.

In this toy problem, we have clearly defined the usage patterns in terms of sets so it is now of interest, whether the proposed method is able to extract these sets.

The algorithm performs the following steps:

1. $\mathcal{L}$ is defined to be the set of all symbols present in the program, that is, in any of the six functions. Each of the sets of symbols S_i is then represented as a vector in an $|\mathcal{L}|$-dimensional vector-space where each dimension is associated with exactly one $l \in \mathcal{L}$. In this way, each function is no longer represented by a set but by a vector in a vector space, which we will refer to as feature space. Vectors are then aligned in columns to form the symbol-function matrix A.
2. Columns of A are centered and PCA is performed to find the two directions of highest variance of the data within this space, the first and second principle component. Furthermore, we calculate two dimensional representations of all functions and symbols. In practice, both the principal components as well as the two dimensional representations of functions and symbols can be obtained by performing a singular value decomposition of A.
3. A scatter plot is then generated to visualize results.

5.1.2 Results

1. Note that the symbol *int*, i.e. the single element of the set $\bigcap_i S_i$ has been projected onto the origin. This is an expected result, since the symbol is present in all of the supplied functions. For that reason, the corresponding dimension in feature space has a variance of zero. This in turn means that the symbol does not contribute to the representation of a function given this data set because it carries no characteristic information, which would allow us to differentiate a function from any of the other functions. This extreme case illustrates the notion of finding descriptive directions in feature space by seeking directions of high variance.
2. Note that the three distinct classes of symbols are clearly visible in the projection: network related symbols and functions have been projected into the upper left, GUI related symbols and functions into the upper right and list-related symbols and functions into the lower right corner.

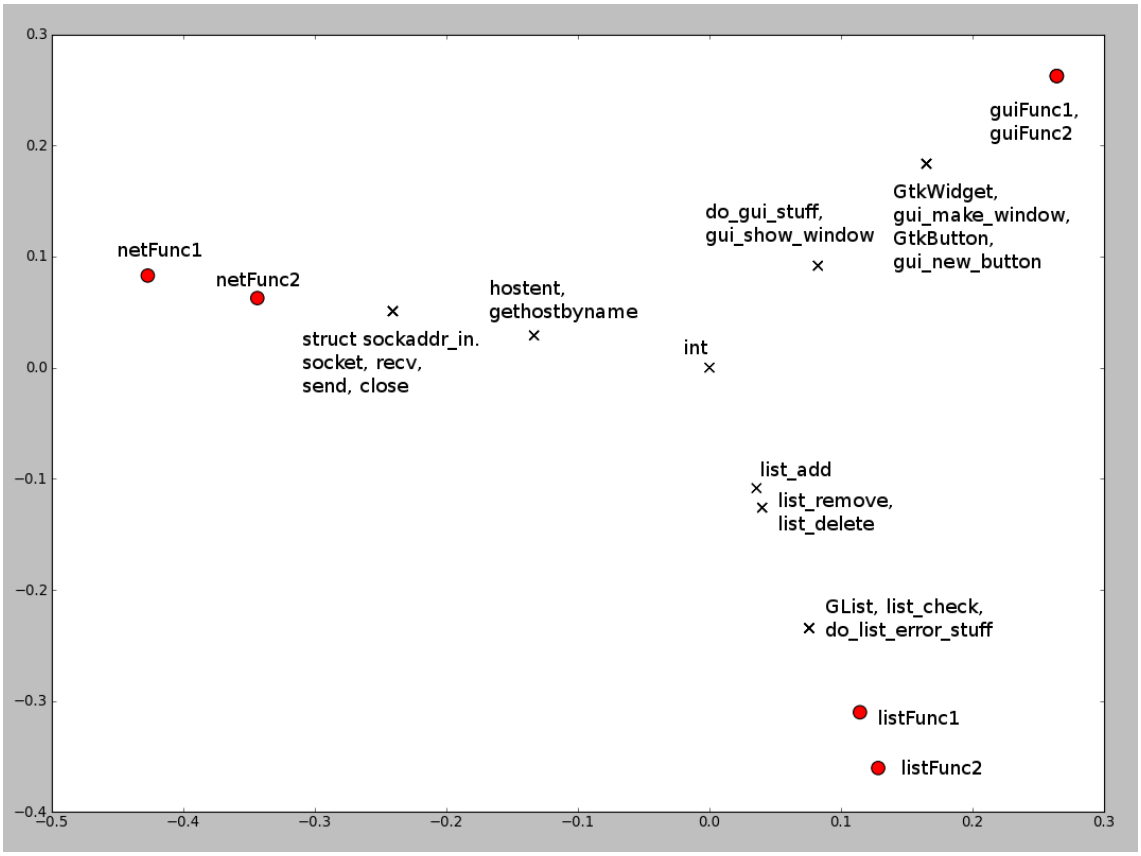


Figure 5.1: Functions (dots) and Symbols (crosses) as projected onto the plane spanned by the first and second principle components

3. Symbols, which have occurred in only a single function e.g. $\{hostent, gethostbyname\}$ are located between the origin and their related API, given that they are more descriptive than symbol *int* but less descriptive than terms occurring in two related functions. This highlights the advantage of measuring similarity within the lower dimensional projection created by PCA as opposed to measuring within feature space directly. In feature space, each of the symbols is associated with a unit vector $\vec{e}_{Symbol_i}$ orthogonal to all unit vectors associated with any of the other symbols, in other words:

$$\vec{e}_{Symbol_i} \cdot \vec{e}_{Symbol_j} = 0 \quad i, j = 1, .., |\mathcal{L}| \\ i \neq j$$

Since a function f not containing the symbol $Symbol_i$ is in general given by a linear combination of a set of unit-vectors $V = \{\vec{e}_{Symbol_j}\}_{j \neq i}$, the scalar product $\vec{e}_{Symbol_i} \cdot f$ is

$$\begin{aligned} \vec{e}_{Symbol_i} \cdot f &= \vec{e}_{Symbol_i} \cdot \sum_{\vec{v} \in V} \vec{v} \\ &= \vec{e}_{Symbol_i} \cdot (\vec{e}_{Symbol_{j1}} + \vec{e}_{Symbol_{j2}} +, \ldots, + \vec{e}_{Symbol_{j|V|}}) \\ &= 0 \end{aligned}$$

In practice, this means that the similarity between a function and a term not contained in the function is always 0 when measured directly in feature space. In contrast, in the lower dimensional approximation of the data created by PCA, a function, which makes use of the same API but does not contain a symbol of interest, is still more similar to the symbol of interest than any of the functions, which make use of a different API. This observation illustrates the link between the notion of generalization and data approximation as performed by PCA. Table 5.1 gives an example: Symbol *hostent* is only present in *netFunc2*. Notice that *netFunc1* is considerably more similar to *hostent* than any function other than *netFunc2*.

4. It may come as no surprise that clustering symbols based on scalar products in the low dimensional space using the K-Means Algorithm with $k = 3$ recovers the sets $S_{Net1} \cup S_{Net2} \cup \{int\}$, $S_{Gui1} \cup S_{Gui2}$ and $S_{List1} \cup S_{List2}$.

netFunc2	netFunc1	listFunc2	listFunc1	guiFunc1	guiFunc2
+0.8788	+0.3468	-0.3156	-0.3256	-0.3428	-0.3428

Table 5.1: Cosine Similarity between function and term *hostent*.

This experiment demonstrates the proposed method's ability to identify the three distinct APIs used in the toy problem and shows how the proposed method projects functions and API symbols into a lower dimensional sub-space in accordance with the dominant API usage patterns present in the code-base. A quantitative evaluation on a real data set follows in section 5.2.

5.1.3 Source Code

```
void netFunc1()
{
        int fd;
        int i = 0;
        struct sockaddr_in in;
        fd = socket(arguments);
        recv(fd, moreArguments);

        if(condition){
                i++;
                send(fd, i, arg);
        }
        send(fd, i, arg);
        close(fd);
}



void guiFunc1(GtkWidget *widget)
{
        int j;
        gui_make_window(widget);
        GtkButton *button;

        button = gui_new_button();
        gui_show_window();
}




void listFunc1(int elem)
{
        GList myList;

        if(! list_check(myList)){
                do_list_error_stuff();
                return;
        }

        list_add(myList, elem);
}
```

```
void netFunc2()
{
        int fd;
        struct sockaddr_in in;
        hostent host;

        fd = socket(arguments);
        recv(fd, moreArguments);
        gethostbyname(host);

        if(condition){
                int i = 0;
                i++;
                send(fd, i, arg);
        }
        close(fd);
}

void guiFunc2(GtkWidget *widget)
{
        gui_make_window(widget);
        GtkButton *myButton;

        button1 = gui_new_button();
        button2 = gui_new_button();
        button3 = gui_new_button();

        for(int i = 10; i != 20; i++)
                do_gui_stuff();
}

void listFunc2(int elem)
{
        GList myList;

        if(! list_check(myList)){
                do_list_error_stuff();
                return;
        }
        list_remove(myList, elem);
        list_delete(myList);
}
```

5.2 Quantitative Evaluation

To quantitatively evaluate the performance of the proposed method, we manually extract distinct known families of functions from popular open source code bases and measure the method's ability to model the similarity between functions within each of the families as well as the dissimilarity between functions of differing families. In particular, it is of interest whether API symbols are suitable features of a function to allow a clear distinction of the different families and whether the application of PCA and TF-IDF can further stabilize the representation of the data. Furthermore, the performance of n-gram kernels to bag-of-words kernels was evaluated.

5.2.1 Setup

Families of related functions were chosen such that members of a family perform functionally similar or equivalent tasks and are taken from the same code base. The functions of a family thus share attack vectors and are likely to be plagued by similar potential vulnerabilities.

In particular, we have chosen the following families of functions:

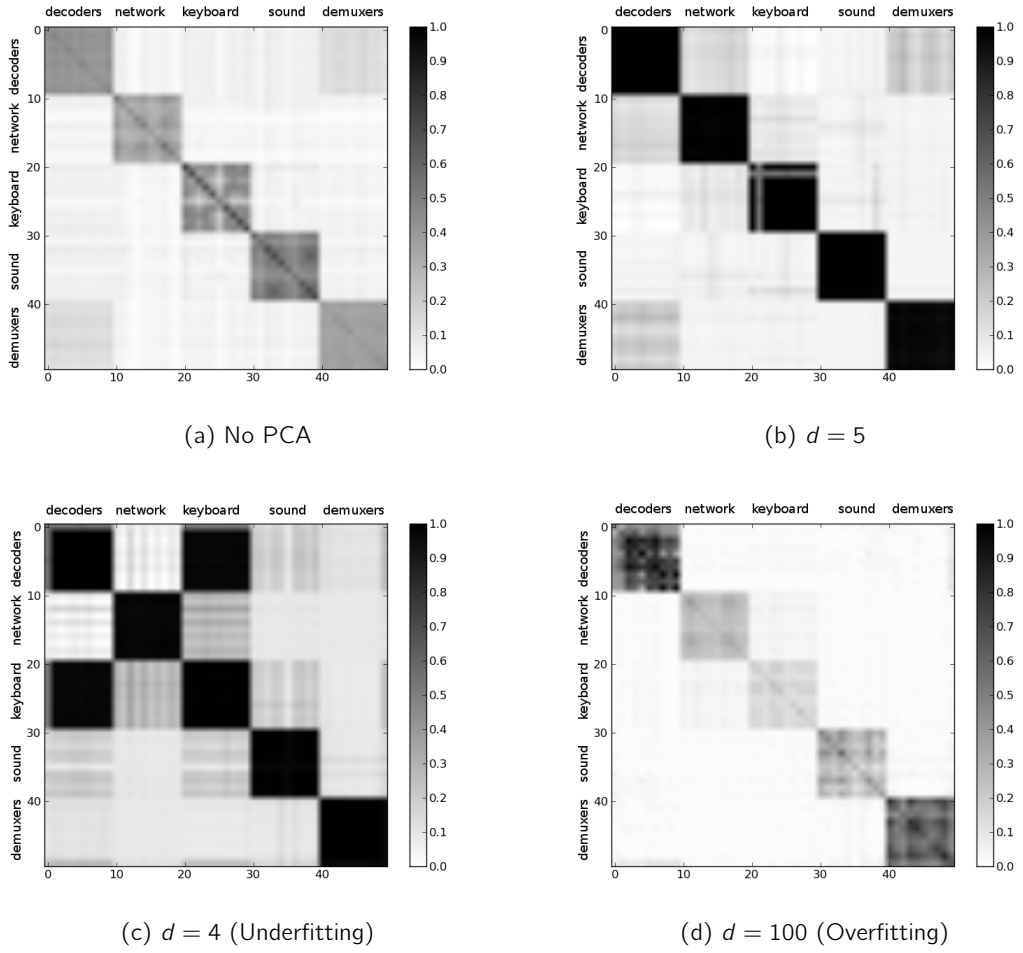
- The *Network* family consisting of packet sending routines defined by Ethernet Drivers of the Linux Kernel.
- The *Keyboard* family consisting of Linux Kernel Keyboard Driver probe routines.
- The *Sound* family consisting of Linux Kernel Sound Driver probe routines.
- The *Demuxer* family consisting of demuxer routines from FFmpeg for different file formats.
- The *Decoder* family consisting of decoder routines from FFmpeg for different media codecs.

Members of each family are first randomly partitioned into subsets such that subsets created from the same family have approximately the same size and each family is divided into ten subsets. We denote the set of all subsets by $\mathcal{S} = \{s_1, \ldots, s_{|\mathcal{S}|}\}$ and define f to be a mapping between a subset and its family. The test error is then measured by the average quadratic distance between the distance function d implicitly calculated by the model and a target function t, i.e. by

$$err = \frac{1}{|\mathcal{S}|^2} \sum_{s_1 \in \mathcal{S}} \sum_{s_2 \in \mathcal{S}} (d(s_1, s_2) - t(s_1, s_2))^2$$

where the target function t is given by

$$t(s_1, s_2) = \begin{cases} 1 & \text{if } f(s1) = f(s2) \\ 0 & \text{otherwise} \end{cases}$$



(a) No PCA

(b) $d = 5$

(c) $d = 4$ (Underfitting)

(d) $d = 100$ (Overfitting)

Figure 5.2: Distance Matrices for different numbers d of eigenvectors used to represent the data. Dark gray values denote high similarities.

and d is the average pairwise cosine distance between elements of s_1 and s_2.

We then measure the development of the test error as the number of dimensions used to represent the data is increased.

5.2.2 Results

1. Figure 5.2a visualizes function similarities as measured directly in the infinite dimensional feature space, i.e. without performing PCA to reduce the dimensionality of the data set. While in this representation, the similarities between functions of the same family as shown on the diagonal of the matrix are on average higher than those between functions of differing families, the difference lacks in significance. Furthermore, a notable variance between similarity scores within a family is observable.

2. Figure 5.2b shows the similarities of functions as represented by their projections onto the first five eigenvectors scaled by their respective eigenvalues. In this lower dimensional

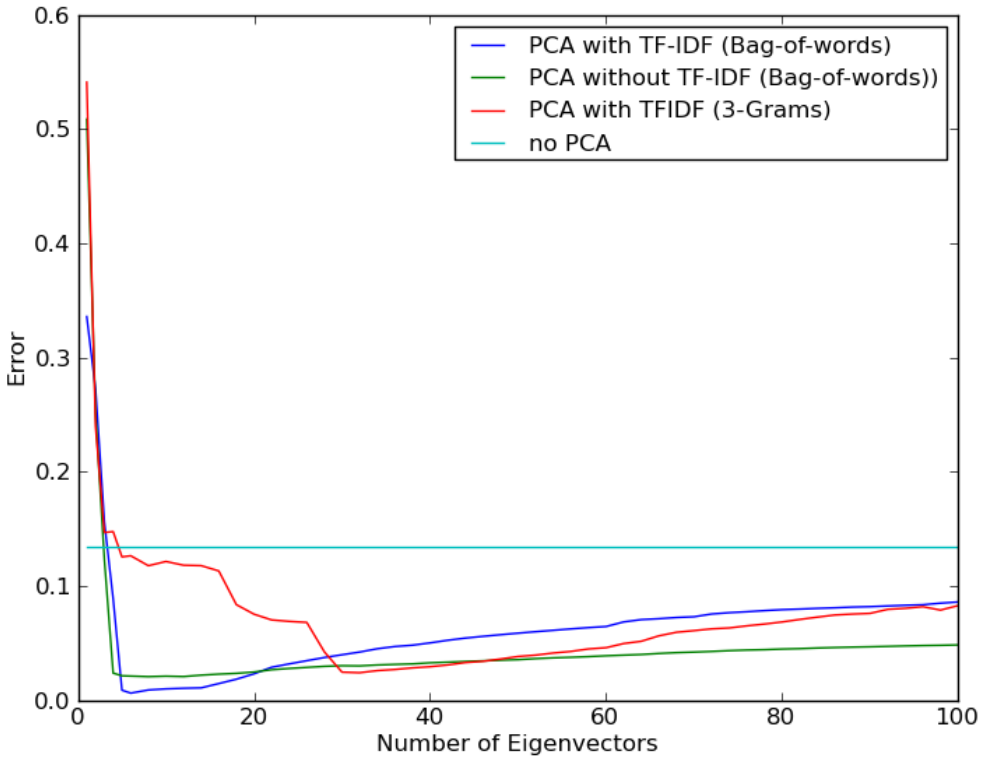
representation, inter-familiar similarities are significantly higher than in the original representation while high distances between functions of differing families have mostly been conserved. This figure demonstrates that the proposed method can be used to remove noise from the representation, thereby yielding a significantly more robust representation of the data.

3. Figures 5.2c and 5.2d demonstrate the importance of choosing a suitable number of dimensions to represent the data. If this number is too low, the data model is not complex enough to represent similarities between members of a family while maintaining dissimilarities to functions of differing families (5.2c). In contrast, if the number of dimensions is too high, the generalization achieved by the data model is low. Consequently, similarities between functions of the same family are not represented adequately (5.2d).

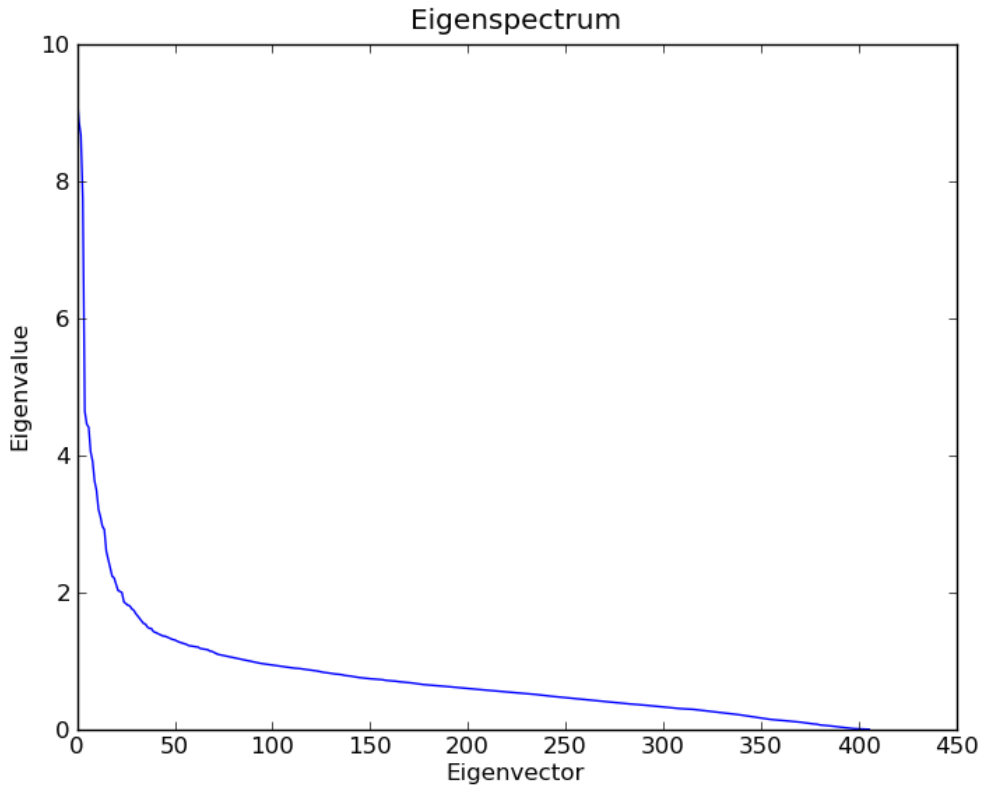
4. Figure 5.3a compares the development of the error as the dimensionality of the representation is increased to the error of the initial high dimensional representation. Given a suitable number of dimensions, *the application of PCA decreases the error by approximately 10 percent*. Furthermore, the graph conforms to the observations made in the previous paragraph; The error first decreases as dimensions are added to the representation until the number of dimensions is high enough to account for the relevant information contained in the data set. The error then slowly increases as an increasing amount of noise is modeled. However, the graph also highlights the fact that the method is robust towards moderate overfitting. Furthermore, the bag of words kernel significantly outperforms the n-gram kernel.

5. Figure 5.3b shows the eigenvalue spectrum. The figure illustrates the fact that the relevant information is contained within a few leading eigenvectors, however, the spectrum is in this case slightly misleading. While the spectrum suggests that about 50 dimensions are necessary to model the data, the experiment shows that far less dimensions are actually required. However, as the error graph shows, choosing about 50 dimensions still produces good results.

In this section, we have demonstrated how mapping functions taken from a real data set to a vector space according to API symbol usage allows loose semantic coupling between functions to be detected by measuring cosine-distances within the corresponding feature space. Furthermore, we have demonstrated that PCA can be used to perform denoising and thus stabilize the representation resulting in a significant error reduction.



(a) Development of the error as the number of eigenvectors used to represent the data is increased.



(b) Eigenvalue spectrum for the bag-of-words representation with TF-IDF, i.e. the eigenvalues sorted in descending order.

CHAPTER 6

Case Studies

Clearly, the success of vulnerability identification by manual source code analysis depends highly on the experience and skill set of the auditor. Therefore the presentation of zero-day vulnerabilities in production code cannot serve the purpose of evaluating the performance of the method. Nonetheless, we feel that a method developed to assist the practitioner must convince the practitioner, and that case studies demonstrating the integration of the method into real code auditing processes are well suited to achieve this. Furthermore, the practitioner will agree that studying the method's ability to re-discover known vulnerabilities is pathological. For this reason, a zero-day vulnerability is presented.

In the typical contemporary execution environments where defense mechanisms such as stack Canaries, ASLR and hardened heap implementations have been put into place to hinder exploitation, *there is a notable difference between potentially exploitable memory corruption vulnerabilities as commonly found by fuzz testing and vulnerabilities, which can provably be exploited to execute arbitrary attacker supplied code.*

Indeed, one of the strengths of manual source code analysis is that the auditor accumulates a thorough understanding of the applications internals as needed to exploit a vulnerability, thereby enabling the auditor to estimate the potential exploitability of the bug even before a crash is triggered. For this reason, we also describe a proof of concept exploit for one of the identified zero-day vulnerabilities and thereby prove that it is indeed a vulnerability.

In section 6.1 we show how we used our tool to discover a vulnerability in FFmpeg similar to a known vulnerability with a single query. An exploit is presented in section 6.1.4. It is then demonstrated how an auditor can use our tool to browse the code base and uncover relevant usage patterns and corresponding functions in section 6.2.

6.1 Vulnerability Extrapolation in FFmpeg

6.1.1 Original Vulnerability

In September 2010, oCert reported a vulnerability in the widely used open-source media decoding library FFmpeg attributed to Cesar Bernardini, which may allow an attacker to execute arbitrary code via crafted *FLIC* media files (CVE-2010-3429)[oCE]. More specifically, an attacker can write arbitrary data to arbitrary locations in memory relative to a pointer on the heap by supplying a crafted *FLIC* video frame.

The vulnerability is contained in the *FLIC* video decoder, specifically in function *flic_decode_frame_8BPP* displayed in listing 6.2, which is called for each frame of an 8 bit per pixel *FLIC* video. The critical write operation is performed on line 33 where the least significant byte of the attacker supplied 16 bit integer *line_packets* is written to a location relative to the heap-based buffer *pixels*. It has been overlooked that the offset is dependent on *y_ptr* and *s->frame.linesize[0]*, both of which are attacker controlled. In fact, due to the loop starting at line 22, it is possible to assign an arbitrary value to *y_ptr* independent of the last value stored in line_packets and no check is performed to verify whether the offset remains within the confined regions of the buffer. It is thus possible for an attacker to write arbitrary bytes to arbitrary locations in memory.

Note that most media players using FFmpeg do not rely on correct file extensions to detect media formats. In fact, even if an incorrect file extension such as the well known *.MP3* is supplied, the correct decoder may still be identified. This adds to the value of the vulnerability for an attacker given that *MP3* is a well known file format while *FLIC* is not.

6.1.2 Extrapolation

As we apply our tool to find functions similar to the known function, an understanding of the scenario, which lead to this vulnerability and a general description of the class this vulnerability belongs to is obtained. Querying for functions similar to the vulnerable function yields a list of other decoding routines, which use similar programming patterns. Upon examination, it becomes clear that the root of the problem is as followed:

Most video file formats define the format of a file header and a format for video and audio frames. In many cases, both the header and the individual frames contain width and height fields, thereby allowing videos with dimensions that change over time to be represented. Throughout FFmpeg's frame decoding routines, it is common practice to choose the number of bytes allocated for frame buffers according to the width and height fields specified in the format header as opposed to the frame dimensions specified by the individual frames. In consequence, the programmer must take care to verify whether the frame dimensions

specified by the frame as well as any offsets into the frame buffer address memory within the allocated buffer.

Score 1	Function Name
1.000000	flic_decode_frame_8BPP (libavcodec/flicvideo.c)
0.964096	flic_decode_frame_15_16BPP (libavcodec/flicvideo.c)
0.826979	lz_unpack (libavcodec/vmdav.c)
0.803331	decode_frame (libavcodec/lcldec.c)
0.796700	raw_encode (libavcodec/rawenc.c)
0.756951	vmdvideo_decode_init (libavcodec/vmdav.c)
0.723750	vmd_decode (libavcodec/vmdav.c)
0.702356	aasc_decode_frame (libavcodec/aasc.c)
0.684610	flic_decode_init (libavcodec/flicvideo.c)
0.665167	decode_format80 (libavcodec/vqavideo.c)
0.664279	targa_decode_rle (libavcodec/targa.c)
0.660454	adpcm_decode_init (libavcodec/adpcm.c)
0.659811	decode_frame (libavcodec/zmbv.c)
0.655338	decode_frame (libavcodec/8bps.c)
0.651587	msrle_decode_8_16_24_32 (libavcodec/msrledec.c)
0.648321	wmavoice_decode_init (libavcodec/wmavoice.c)
0.646872	get_quant (libavcodec/nuv.c)
0.641871	MP3lame_encode_frame (libavcodec/libmp3lame.c)
0.641642	mpegts_write_section (libavformat/mpegtsenc.c)
0.634922	tgv_decode_frame (libavcodec/eatgv.c)

Table 6.1: Top 20 of a total of 6778 functions, ranked by cosine distance to flic_decode_frame_8BPP.

Table 6.1 summarizes the result of comparing all 6778 functions recognized by our tool to the query function *flic_decode_frame_8BPP*. The table contains the twenty most similar functions as well as their corresponding similarity scores. First, note that a similar bug was fixed in *flic_decode_frame_15_16BPP* and that our tool reports a similarity score of 96% for the two functions. Extrapolation on the spot, that is the correction of similar bugs within the scope of a file, is commonly achieved, as has been in this case. However, a similar vulnerability in function *vmd_decode* shown in listing 6.3 from a different source file was not spotted.

6.1.3 Zero-Day Vulnerability

As is the case for *FLIC* video frames, for *VMD* video frames, allocation of the frame buffer is performed according to the frame dimensions specified in the format header. This takes place in function *vmdvideo_decode_frame* by calling *avctx->get_buffer* on line 16. The vulnerable function *vmd_decode* responsible for storing the decoded frame data in the frame buffer is then called on line 21.

On line 8 to 11 *vmd_decode* now reads the frame dimensions specified by the individual frame as well as an x and y offset from the attacker supplied file. Based on this information, an offset

into the frame buffer is now calculated on line 33 and the function fails to validate whether the given offset references a location within the buffer. Therefore, as attacker supplied data is copied to the specified location on line 42, by choosing an offset outside of the buffer, an attacker can corrupt memory.

Both vulnerabilities conform to the following general description:

1. Frame dimensions are read from the attacker supplied data source by use of the *AV_RL* family of functions.
2. An offset into the pixel buffer provided by a video context structure is calculated based on the attacker supplied frame dimensions and the code fails to verify whether the offset is larger than the size of the buffer.
3. Attacker supplied data is written to the location defined by the pixel buffer and the calculated offset.

This example demonstrates that API usage patterns are a good choice for modeling functions from a security perspective. In fact, the *AV_RL* family of functions correspond to the data sources controlled by an attacker and it comes as no surprise that similar data sources are used for similar purposes. The corresponding sinks are provided by the low level routines from the C Standard Library and the application specific error handling API *av_log*. It is the combination of these API symbols, which provides a good fingerprint for a set of functions within FFmpeg. Also note that the two functions are *not* copy paste clones.

To verify whether this bug is indeed a vulnerability, we must proof that it is exploitable. Given that an attacker has no benefit from crashing a victim's media player, if this bug does not allow for attacker supplied code to be executed, it is not a vulnerability.

We describe a proof of concept exploit in the following section.

6.1.4 Exploit

To exploit the presented bug, we first observe that after calling *vmd_decode*, *vmd-video_decode_frame* releases the previous frame buffer by first swapping the current and the previous frame and then calling *avctx->release_buffer*, passing the current frame as the second argument on line 29. The *AvCodecContext* structure *avctx* therefore contains a pointer to a function, which when overwritten, allows an attacker to redirect program execution.

Debugging the target program reveals that *avctx* is located on the program heap just like the frame buffer. However, since *avctx* is allocated on program initialization, it is likely to be located at a lower address. Keep in mind, however, that due to multithreading, we can in general not assume that the frame buffer and the *avctx* structure are separated by a constant offset.

To reliably overwrite *avctx* and in particular the *release_buffer* pointer, we must therefore be able to specify negative offsets, given that the frame buffer may be located at a higher address than the *avctx*. Note that since both items are on the heap and an address relative to the frame buffer can be specified, *ASRL* does not in this case complicate exploitation. However, if ASRL had not been present, a more stable exploit could have been developed by patching the global offset table.

As shown on line 33, the offset is specified by the two 16 bit integers *frame_x* and *frame_y* as well as *s->frame.linesize[0]*, which we can influence by means of the width field in the format header. However, all of these variables can only be set to a range between 0 and 65535. Therefore, we cannot specify offsets in the interval $[-65535; -1]$ with a single frame.

Furthermore, using a single frame, whenever *frame_width* differs from the *avctx->width*, i.e. the width field specified in the format header, *vmd_decode* copies the previous frame to the current frame on line 23. If no previous frame exists, this leads to a null pointer dereference and an immediate program crash. However, *frame_width* is the parameter needed to control the amount of data to be copied, which limits us to copying *avctx->width* bytes of data if only a single frame is used to trigger the vulnerability.

To bypass these limitations, the bug is exploited using *two frames*: The first frame contains the sign inverted desired offset in *frame_x*. This value is then stored in *s->x_off* on line 15 and subtracted from *frame_x* on line 18 such that the following block is not executed. *frame_y* is set to 0 and *frame_width* is set to a value smaller than the size of the allocated frame buffer such that the copy operation on line 42 does not overflow the buffer. The first frame thus does not trigger the vulnerability. Its purpose is to allocate *s->prev_frame.data[0]* and save the desired offset in *s->x_off*.

The second frame now uses a *frame_x* and *frame_y* value of 0 such that line 15 and 16 are *not* executed. On line 18, the value of *s->x_off* stored by the previous frame is now subtracted from *frame_x*, thereby resulting in an underflow. *frame_x* has now taken on a negative value as desired. The copy operation on line 42 now copies an amount of attacker supplied data specified by *frame_width* to a location before the buffer. We exploit this to overwrite large portions of the heap before the buffer and in particular the *release_buffer* pointer.

Once control over the flow of execution has been obtained, a suitable sequence of instructions at a constant address must be identified that can be leveraged to execute arbitrary attacker supplied code. At this point it is notable that while implementations of ASLR for recent Linux kernels randomize the heap, stack and mmap base, the location of the base program image remains constant. Luckily, for an attacker, *mplayer* contains the following sequence of instructions at a constant address:

```
0x080cc5c2 :    mov    %eax,(%esp)
0x080cc5c5 :    call   0x809481c <system@plt>
```

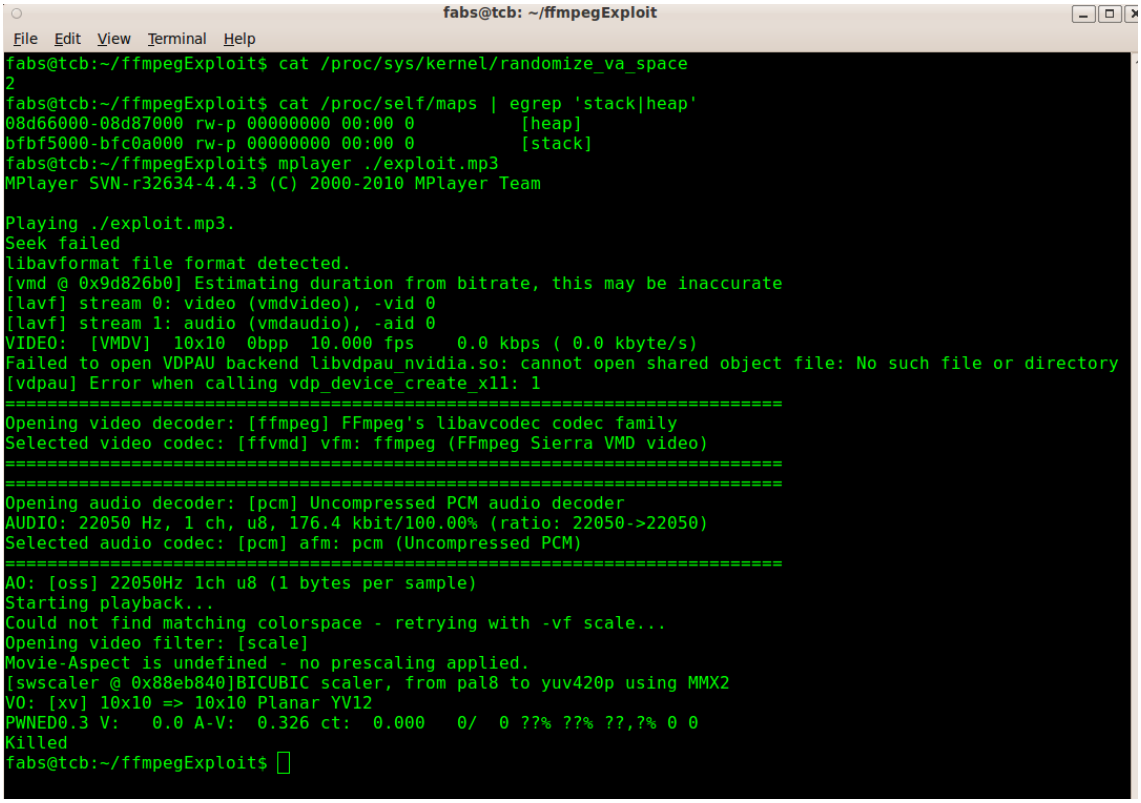


Figure 6.1: Terminal window showing running proof of concept exploit. ASRL is enabled and stack and heap are non-executable. The exploit prints the message "PWNED" and kills the mplayer process.

As seen on line 29, *release_buffer* receives *avctx* as its first argument. To accomplish this, *avctx* was first moved into the register %eax and then pushed onto the stack, therefore, as the sequence of instructions is invoked, the shell commands saved at *avctx* are executed. Since we have been able to overwrite *avctx->release_buffer*, it is also possible to overwrite *avctx*. To exploit the issue, we therefore overwrite the *avctx* structure such that it begins with the shell commands to be executed and replaces *avctx->release_buffer* with the address of the instruction sequence presented. Note that *avctx* and *avctx->release_buffer* are 260 bytes apart, leaving enough room for shell commands.

It is noteworthy that to stabilize this exploit, such that it runs across various video players using FFmpeg in a variety of different versions, the attacker must gain a more fine grained control over the heap state. However, this remarkably simple exploit still proofs that the vulnerability can indeed be exploited on recent Linux systems despite non-executable stack and heap regions as well as ASLR.

6.2 API Discovery in FFmpeg

By calculating dot products between the vector associated with *flic_decode_frame_8BPP* v_1 and all symbol vectors u_i, we can uncover the broader API as extracted by our tool shown in table 6.2b. Note that only symbol vectors with a sufficient length, i.e. such symbol vectors, which account for a large amount of variance within the data set, are possible candidates for the broader API since $v_1 \cdot u_i = \|v_1\|\|u_i\| \cdot cos(\angle(v_1, u_i))$, thus the similarity as measured by dot products is proportional to the length of the vectors.

Notice that the broader API as shown in table 6.2b contains all symbols mentioned in the general description of the vulnerability pattern as given in the previous section. In particular, the *AV_RL* as well as the *AV_RB* family of sources as well as the sinks *memcpy*, *memset* and *av_log* are present regardless of the fact that not all of these symbols are actually contained in *flic_decode_frame_8BPP*.

In contrast, the symbol *FlicDecodeContext* is not contained within the broader API of the function as uncovered by our tool due to its insufficient length. It therefore does not play an important role in the representation of the data set and merely corresponds to a small deviation. However, it has been projected at an angle corresponding to that of the broader API it accompanies. By querying our tool for symbols of equal angle, e.g. by calculating cosine distances between the vector *FlicDecodeContext* and all other symbol vectors, we can uncover other small deviations to the broader API as shown in table 6.2a.

The results shown in table 6.2a correspond well to intuition: The broader API is accompanied by numerous decoding contexts, all of which are themselves only present in very few functions. These symbols are noise, which make a function unique while the broader API represents the overall signal shared between the functions.

We have now discovered that FFmpeg contains a set of frame decoding routines similar to *flic_decode_frame_8BPP*, which share a common API. However, the view obtained of this API is influenced to a considerable degree by the specific properties of *flic_decode_frame_8BPP*. To decrease this influence and arrive at a more general view of the API shared by frame decoding routines, the center of mass

$$m = \frac{1}{|\mathcal{S}|} \sum_{v \in \mathcal{S}} v$$

of a set $\mathcal{S}$ of frame decoding routines can be used as a query. Our tool therefore offers the ability of calculating the API, which best matches a set of functions specified by a user. The center of mass m of all function vectors associated with the function name *decode_frame* (a total of 50 functions) has been calculated. By comparing m to all symbol vectors using dot products, the API displayed in table 6.3b was uncovered.

The inverse operation, i.e. finding the set of functions, which best match a given set of API symbols is also supported. Consider for example that an auditor is interested in finding all

Score	Symbol
1.00	FlicDecodeContext
0.88,	EightBpsContext * const
0.88,	AnmContext
0.87,	TiffContext * const
0.87,	IdcinContext
0.86,	free_bundles
0.86,	C93DecoderContext * const
0.86,	SgiState * const
0.86,	VCR1Context
0.86,	ff_lzw_decode_close
0.84,	release_buffer
0.81,	BinkContext * const
0.80,	VmdVideoContext
0.80,	LibOpenJPEGContext
0.80,	MsrleContext
0.77,	TMVContext
0.77,	DVVideoContext
0.76,	Msvideo1Context
0.76,	ff_ivi_free_buffers
0.75,	VqaContext
0.74,	KmvcContext *const
0.73,	inflateEnd
0.73,	Rl2Context
0.71,	CmvContext
0.71,	CyuvDecodeContext
0.69,	CamtasiaContext * const
0.69,	AuraDecodeContext
0.67,	SmackVContext * const
0.67,	SmcContext
0.67,	AascContext
0.66,	RpzaContext
0.66,	ZmbvContext * const
0.66,	BethsoftvidContext
0.66,	FrapsContext
0.65,	VBDecContext
0.64,	LclDecContext * const
0.64,	IffContext
0.63,	MmContext

(a)

API Symbol	Score
AV_RL32	2.89
unsigned int	2.43
unsigned char	2.26
void	2.11
AVCodecContext	2.05
memset	1.81
const uint8_t	1.73
memcpy	1.47
av_log	1.30
AV_RL16	1.15
const unsigned char	0.89
AVProbeData	0.84
uint8_t	0.54
CHECK_STREAM_PTR	0.50
release_buffer	0.41
AVPacket	0.35
av_malloc	0.29
QtrleContext	0.28
get_buffer	0.27
CHECK_PIXEL_PTR	0.25
avcodec_alloc_frame	0.25
IpvideoContext	0.24
SeqVideoContext	0.24
init_get_bits	0.22
uint32_t	0.22
DSPContext	0.21
AVERROR	0.21
av_clip_uint8	0.20
AV_RB16	0.19
AV_RL64	0.19
AV_RB32	0.18
av_realloc	0.18
dprintf	0.18
AVFrame	0.17
FlicDecodeContext	
int	
signed short	
reget_buffer	

(b)

Table 6.2: Table 6.2a shows the cosine distances of symbols to *FlicDecodeContext*. Decoding Contexts are noise terms projected at an angle corresponding to the broader API they accompany. By measuring cosine distances between a vector associated with a decoding context and all other symbol vectors, they can be extracted. Decoding Contexts are marked in grey. Table 6.2b shows the broader API of *flic_decode_frame_8BPP* extracted by calculation of dot products to symbol vectors. Symbols present in the function are marked in grey. Symbols, which are not part of the broader API but are present in the function have been appended to the table.

Score	Function Name
0.92	libavcodec/ulti.c ulti_decode_frame
0.91	libavcodec/loco.c decode_frame
0.90	libavcodec/gifdec.c gif_decode_frame
0.90	libavcodec/qpeg.c decode_frame
0.90	libavcodec/cscd.c decode_frame
0.90	libavcodec/ptx.c ptx_decode_frame
0.90	libavcodec/vcr1.c decode_frame
0.90	libavcodec/indeo5.c decode_frame
0.90	libavcodec/dv.c dvvideo_decode_frame
0.89	libavcodec/cljr.c decode_frame
0.89	libavcodec/tiff.c decode_frame
0.89	libavcodec/indeo3.c indeo3_decode_frame
0.89	libavcodec/vqavideo.c vqa_decode_frame
0.89	libavcodec/indeo2.c ir2_decode_frame
0.88	libavcodec/rl2.c rl2_decode_frame
0.88	libavcodec/vmdav.c vmdvideo_decode_frame
0.88	libavcodec/idcinvideo.c idcin_decode_frame
0.88	libavcodec/pcx.c pcx_decode_frame
0.88	libavcodec/frwu.c decode_frame
0.88	libavcodec/sgidec.c decode_frame
0.88	libavcodec/aura.c/aura_decode_frame
0.87	libavcodec/nuv.c/decode_frame
0.87	libavcodec/dnxhddec.c/dnxhd_decode_frame
0.87	libavcodec/mimic.c/mimic_decode_frame
0.86	libavcodec/kmvc.c/decode_frame
0.86	libavcodec/cdgraphics.c/cdg_decode_frame
0.86	libavcodec/dxa.c/decode_frame
0.86	libavcodec/ffv1.c/decode_frame
0.86	libavcodec/vb.c/decode_frame
0.84	libavcodec/mdec.c/decode_frame

(a)

API Symbol	Score
AVCodecContext	98.018
release_buffer	91.86
void	87.57
AVPacket	69.11
const uint8_t	67.77
av_log	54.94
AVFrame	46.97
uint8_t	43.86
memcpy	34.15
uint32_t	27.02
AV_RL32	25.51
unsigned int	23.03
uint16_t	21.26
memset	21.13
get_buffer	19.00
GetBitContext	18.72
AVERROR	18.63
get_bits	17.08
av_free	16.33
unsigned char	16.26
AVFormatContext	15.40
av_freep	14.93
av_get_packet	14.58
av_malloc	14.46
AV_RL16	13.62
AV_RB32	12.53
get_bits1	11.79
FFMIN	11.77
const uint32_t	11.617
const int	11.39

(b)

Table 6.3: Table 6.3a shows functions similar to the function vector created by calculating the center of mass of all vectors associated with functions named exactly *decode_frame*.Table 6.3b shows an approximation to the API of frame decoding routines in FFmpeg extracted by calculating dot products of symbol vectors to the center of mass of all vectors associated with functions named exactly *decode_frame*.

Score	Function Name
0.92	matroskadec.c matroska_convert_tag
0.92	sdp.c sdp_get_address
0.92	avfiltergraph.c avfilter_graph_get_filter
0.90	rtsp.c rtsp_parse_transport
0.90	utils.c find_info_tag
0.89	rtsp.c make_setup_request
0.87	movenc.c mov_write_string_metadata
0.84	nutdec.c set_disposition_bits
0.84	tests/rotozoom.c init_demo
0.82	libavutil/error.c av_strerror
0.82	opt.c hexchar2int
0.80	httpauth.c choose_qop
0.80	pnm.c ff_pnm_decode_header
0.80	ffserver.c socket_open_listen
0.80	rdt.c ff_rdt_subscribe_rule
0.80	ffserver.c http_send_too_busy_reply
0.79	audioconvert.c avcodec_sample_fmt_string
0.79	sdp.c sdp_write_header
0.79	ffmpeg.c opt_vstats
0.79	httpauth.c handle_digest_update
0.79	httpauth.c handle_digest_params
0.79	pnm.c pnm_get
0.78	httpauth.c handle_basic_params
0.78	sdp.c sdp_write_address
0.77	ffserver.c ctime1
0.77	ffserver.c find_rtp_session_with_url
0.76	ffserver.c get_arg
0.76	rtpproto.c build_udp_url
0.76	img2.c find_image_range
0.76	utils.c av_match_ext

(a)

Symbol	Score
char	69.12
const char	46.98
strcmp	33.82
snprintf	19.17
exit	16.78
strtol	15.93
fprintf	12.23
strlen	10.47
get_byte	8.28
ByteIOContext	7.31
av_strlcpy	6.61
HTTPContext	6.55
enum AVMediaType	6.11
FF_ARRAY_ELEMS	5.55
av_log	5.44
AVFilterGraph	5.13
strncmp	5.05
av_strlcatf	4.94
ADPCMChannelStatus	4.76
AVCodec	4.68
AVFormatContext	4.59
AVFilterInOut	4.37
skip_spaces	4.35
enum PixelFormat	3.99
strchr	3.87
p	3.84
av_metadata_set2	3.84
av_get_frame_filename	3.64
av_free	3.46
isspace	3.4

(b)

Table 6.4: Table 6.4b shows the API extracted by calculating dot products to the center of mass of the symbol vectors associated with *char*, *strcmp* and *strcpy*. Table 6.4a shows functions similar to this API as given by cosine distances.

functions in the code base, which perform string operations. An auditor may construct a vector representing the desired API by calculating the center of mass of a set of symbol vectors. The cosine distance of m to all function vectors can then be calculated to determine the functions best matching the constructed API. Table 6.4a shows can example.

6.3 Code Listings

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

Figure 6.2: Function reported to be vulnerable by oCert.

```
1  static void vmd_decode(VmdVideoContext *s)
2  {
3     [..]
4      int frame_x, frame_y;
5      int frame_width, frame_height;
6      int dp_size;
7
8      frame_x = AV_RL16(&s->buf[6]);
9      frame_y = AV_RL16(&s->buf[8]);
10     frame_width = AV_RL16(&s->buf[10]) - frame_x + 1;
11     frame_height = AV_RL16(&s->buf[12]) - frame_y + 1;
12
13     if ((frame_width == s->avctx->width && frame_height == s->avctx->height) &&
14         (frame_x || frame_y)) {
15         s->x_off = frame_x;
16         s->y_off = frame_y;
17     }
18     frame_x -= s->x_off;
19     frame_y -= s->y_off;
20     [..]
21     if (frame_x || frame_y || (frame_width != s->avctx->width) ||
22         (frame_height != s->avctx->height)) {
23         memcpy(s->frame.data[0], s->prev_frame.data[0],
24             s->avctx->height * s->frame.linesize[0]);
25     }
26     [..]
27     if (s->size >= 0) {
28         /* originally UnpackFrame in VAG's code */
29         pb = p;
30         meth = *pb++;
31         [..]
32
33         dp = &s->frame.data[0][frame_y * s->frame.linesize[0] + frame_x];
34         dp_size = s->frame.linesize[0] * s->avctx->height;
35         pp = &s->prev_frame.data[0][frame_y * s->prev_frame.linesize[0] + frame_x];
36
37         switch (meth) {
38         [..]
39         case 2:
40             for (i = 0; i < frame_height; i++) {
41
42                 memcpy(dp, pb, frame_width);
43                 pb += frame_width;
44                 dp += s->frame.linesize[0];
45                 pp += s->prev_frame.linesize[0];
46             }
47             break;
48                 [..]
49         }
50     }
51 }
52
```

Figure 6.3: Function found vulnerable by vulnerability extrapolation.

```
1  static int vmdvideo_decode_frame(AVCodecContext *avctx,
2                                   void *data, int *data_size,
3                                   AVPacket *avpkt)
4  {
5      const uint8_t *buf = avpkt->data;
6      int buf_size = avpkt->size;
7      VmdVideoContext *s = avctx->priv_data;
8
9      s->buf = buf;
10     s->size = buf_size;
11
12     if (buf_size < 16)
13         return buf_size;
14
15     s->frame.reference = 1;
16     if (avctx->get_buffer(avctx, &s->frame)) {
17         av_log(s->avctx, AV_LOG_ERROR, "VMD Video: get_buffer() failed\n");
18         return -1;
19     }
20
21     vmd_decode(s);
22
23     /* make the palette available on the way out */
24     memcpy(s->frame.data[1], s->palette, PALETTE_COUNT * 4);
25
26     /* shuffle frames */
27     FFSWAP(AVFrame, s->frame, s->prev_frame);
28     if (s->frame.data[0])
29          avctx->release_buffer(avctx, &s->frame);
30
31     *data_size = sizeof(AVFrame);
32     *(AVFrame*)data = s->prev_frame;
33
34     /* report that the buffer was completely consumed */
35     return buf_size;
36 }
```

Figure 6.4: Caller of vulnerable function *vmdDecode*.

CHAPTER 7

Conclusion

In this work, we have introduced a method, which allows an auditor to quickly discover the most dominant API usage patterns within an application and browse through occurrences to understand how the pattern is put to work and study its subtleties. Furthermore, we have demonstrated on real production code that once a vulnerability is known, similar zero-day vulnerabilities can be identified by cycling through the most similar functions determined by our method. This illustrates an important point: Fixing single vulnerabilities *without performing sufficient extrapolation* may be contra-productive given that it provides attackers with information, which may be used to identify similar zero-day vulnerabilities. Vulnerability extrapolation is therefore an important step in securing software and in this work, we offer a method to assist in this process.

From a more theoretical point of view, it has been shown that unsupervised machine learning algorithms as applied in natural language processing can be used to obtain a representation of source code as a composition of the most dominant API usage patterns present in the application and an additive noise term. In an experiment, we then proceeded to demonstrate that a simple representation of functions by the API symbol contained within each function can already give indications for groups of semantically related functions but results are subject to a considerable amount of noise. It is then by applying Principal Component Analysis that we introduce generalization and thus perform denoising, which significantly improves the method's ability to find semantically related groups of functions.

We see the following ways to directly improve upon the research presented.

1. While vectors representing functions in feature space are sparse, once principal component analysis has been applied, vectors are dense. The associated memory usage, particularly during matrix factorization may not be tolerable for large code-bases, which

require a high number of dimensions to adequately model its complexity. To solve this problem, one may consider to use algorithms such as Sparse Principal Component Analysis [ZHT06] or Non-Negative Matrix Factorization [LS01] as a drop in replacement for Principal Component Analysis.

2. It may be possible to apply this work to binary analysis. In this case, while type information is not directly accessible, it may be sufficient to model functions purely in terms of ingoing and out-going function calls. In this context, it may also make sense to introduce structural features of functions into the representation.

3. Semi supervised versions of Principal Component Analysis [ZZC07] may be applicable, such that the system can improve its representation based on auditor supplied knowledge. Ideally, the representation can be efficiently regenerated whenever the auditor has discovered a new structural property of the code base.

List of Figures

List of Tables

Bibliography

[AHLR07] C. Anley, J. Heasman, F. Lindner, and G. Richarte. *The Shellcoder's Handbook*, chapter 18. Wiley Pub, 2007.

[ano01] anonymous. Once upon a free()... *Phrack 57*, 2001.

[BCH+09] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, 2009.

[BCJ+07] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. RICH: Automatically protecting against integer-based vulnerabilities. In *NDSS (Symp. on Network and Distributed System Security)*. Citeseer, 2007.

[Bis07] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1st ed. 2006. corr. 2nd printing edition, October 2007.

[BKS+07] Stefan Bellon, Rainer Koschke, Ieee Computer Society, Giuliano Antoniol, Jens Krinke, Ieee Computer Society, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE TSE*, 33:577–591, 2007.

[ble02] blexim. Basic integer overflows. *Phrack 60*, 2002.

[Blo06] Joshua Bloch. How to design a good api and why it matters. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 506–507, New York, NY, USA, 2006. ACM.

[BOA+07] Michael Bailey, Jon Oberheide, Jon Andersen, Zhuoqing Morley Mao, Farnam Jahanian, and Jose Nazario. Automated Classification and Analysis of Internet

Malware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID '07)*, pages 178–197, Gold Coast, Australia, September 2007.

[Bra07] S. Bratus. What hackers learn that the rest of us don't. *IEEE Security and Privacy*, 2007.

[Bs06] C.M. Bishop and SpringerLink (Online service). *Pattern recognition and machine learning*, volume 4. Springer New York, 2006.

[BSD10] BSDaemon. Dynamic program analysis and software exploitation. *Phrack 67*, 2010.

[BYM+98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna, and Lorraine Bier. Clone detection using abstract syntax trees, 1998.

[Cui07] Weidong Cui. Discoverer: Automatic protocol reverse engineering from network traces. In *In Proceedings of the 16th USENIX Security Symposium (Security'07*, 2007.

[DDF+90] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[DEP07] J. DeMott, R.J. Enbody, and W.F. Punch. Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing. *BlackHat and Defcon*, 2007.

[DHS01] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2 edition, November 2001.

[DMS06] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006.

[DSD09] Mark Dowd, Ryan Smith, and David Dewey. Attacking interoperability, 2009.

[ECC00] Dawson Engler, Benjamin Chelf, and Andy Chou. Checking system rules using system-specific, programmer-written compiler extensions. pages 1–16, 2000.

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. pages 57–72, 2001.

[Enu] Common Weakness Enumeration. Cwe-170: Improper null termination. `http://cwe.mitre.org/data/definitions/170.html`.

[FOR] An HP Company FORTIFY. Rough auditing tool for security. `https://www.fortify.com/ssa-elements/threat-intelligence/rats.html`.

[Fou] Free Software Foundation. Bison gnu parser generator. `http://www.gnu.org/software/bison/`.

[Hot33] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417–441, 1933.

[II07] Kenneth L. Ingham and Hajime Inoue. Comparing anomaly detection techniques for http. In *Proceedings of the 10th international conference on Recent advances in intrusion detection*, RAID'07, pages 42–62, Berlin, Heidelberg, 2007. Springer-Verlag.

[JKK06] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–263. IEEE, 2006.

[KDM+96] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3:108, 1996.

[Kin] Brad King. Gcc-xml website. `http://www.gccxml.org/HTML/Index.html`.

[KKI02] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, pages 654–670, 2002.

[KL99] G. KNAPEN and E. LAGU. B., DAGENAIS, M., AND MERLO, E. 1999. Parsing C++ despite missing declarations. In *Proc. IWPC*, pages 114–122, 1999.

[KLA+04] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. *The shellcoder's handbook: discovering and exploiting security holes*, chapter 16. John Wiley & Sons, 2004.

[Kle10] Tobias Klein. *Aus dem Tagebuch eines Bughunters*. 2010.

[KM06] J.Z. Kolter and M.A. Maloof. Learning to detect and classify malicious executables in the wild. *The Journal of Machine Learning Research*, 7:2744, 2006.

[Kop97] R. Koppler. A systematic approach to fuzzy parsing. *Software: Practice and Experience*, 27(6):637–649, 1997.

[KV03] Christopher Kruegel and Giovanni Vigna. Anomaly Detection of Web-based Attacks. In *10th ACM Conference on Computer and Communications Security (CCS)*, 2003.

[Lab] MIT Media Lab. Common sense computing initiative - divisi. `http://csc.media.mit.edu/divisi`.

[Liv05] Benjamin Livshits. Dynamine: Finding common error patterns by mining software revision histories. In *In ESEC/FSE*, pages 296–305. ACM Press, 2005.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *SSYM'05: Proceedings of the 14th*

conference on USENIX Security Symposium, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[LLMZ04] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation-Volume 6*, page 20. USENIX Association, 2004.

[LS01] D.D. Lee and H.S. Seung. Algorithms for non-negative matrix factorization. *Advances in neural information processing systems*, 13, 2001.

[LT] Christopher Li Linus Torvals, Josh Triplett. Sparse - a semantic parser for c. `http://sparse.wiki.kernel.org/`.

[LZ05] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 306–315, New York, NY, USA, 2005. ACM.

[McC76] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2:308–320, July 1976.

[McP] Scott McPeak. Elkhound: A glr parser generator. `http://scottmcpeak.com/elkhound/`.

[MM01] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code, 2001.

[Moo01] Leon Moonen. Generating robust parsers using island grammars. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 13, Washington, DC, USA, 2001. IEEE Computer Society.

[NKS05] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *2005 IEEE Symposium on Security and Privacy*, pages 226–241, 2005.

[NS05] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Citeseer, 2005.

[oCE] oCERT. 2010-004 ffmpeg/libavcodec arbitrary offset dereference. `http://www.ocert.org/advisories/ocert-2010-004.html`.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack 49*, 1996.

[Par] Terence Parr. Antlrv3 parser generator. `http://www.antlr.org/`.

[PDL+06] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. In *2006 IEEE Symposium on Security and Privacy*, page 15, 2006.

[PPQ95] Terence J Parr, T. J. Parr, and R W Quong. Antlr: A predicated-ll(k) parser generator, 1995.

[RHW+08] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick DÃÂ¼ssel, and Pavel Laskov. Learning and Classification of Malware Behavior. In Diego Zamboni, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, chapter 6, pages 108–125–125. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[Rie09] K. Rieck. *Machine learning for application-layer intrusion detection*. PhD thesis, 2009.

[Roh] Doug Rohde. Svdlibc - doug rohde's svd c library. `http://tedlab.mit.edu/~dr/SVDLIBC/`.

[SB88] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. In *INFORMATION PROCESSING AND MANAGEMENT*, pages 513–523, 1988.

[Scu01] Team TESO Scut. Exploiting format string vulnerabilities version 1.2. 2001.

[SECZ07] S. Sparks, S. Embleton, R. Cunningham, and C. Zou. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 477–486. IEEE, 2007.

[Sot09] Alexander Sotirov. Bypassing memory protections:the future of exploitation, 2009.

[SPP+04] H. Shacham, M. Page, B. Pfaff, E.J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[SRR07] Sören Sonnenburg, Gunnar Rätsch, and Konrad Rieck. Large scale learning with string kernels. In Léon Bottou, Olivier Chapelle, Dennis DeCoste, and Jason Weston, editors, *Large Scale Kernel Machines*, pages 73–103. MIT Press, Cambridge, MA., 2007.

[SSM98] B. Schölkopf, A. Smola, and K.R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319, 1998.

[STC04] J. Shawe-Taylor and N. Cristianini. *Kernel methods for pattern analysis*. Cambridge Univ Pr, 2004.

[TP02] M.A. Turk and A.P. Pentland. Face recognition using eigenfaces. In *Computer Vision and Pattern Recognition, 1991. Proceedings CVPR'91., IEEE Computer Society Conference on*, pages 586–591. IEEE, 2002.

[twi00] twitch. Taking advantage of non-terminated adjacent memory spaces. *Phrack 56*, 2000.

[Van10] Zero-sized heap allocations vulnerability analysis. In *4th USENIX Workshop on Offensive Technologies (WOOT 10)*, 2010.

[VBKM02] J. Viega, JT Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*, pages 257–267. IEEE, 2002.

[WCKK08] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08*, 2008.

[Whe] David A. Wheeler. Flawfinder. `http://www.dwheeler.com/flawfinder/`.

[WHM05] Chadd C. Williams, Jeffrey K. Hollingsworth, and Senior Member. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31:466–480, 2005.

[WS04] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. pages 203–222, 2004.

[WWLZ09] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*, 2009.

[ZHT06] H. Zou, T. Hastie, and R. Tibshirani. Sparse principal component analysis. *Journal of computational and graphical statistics*, 15(2):265–286, 2006.

[ZZC07] D. Zhang, Z.H. Zhou, and S. Chen. Semi-supervised dimensionality reduction. In *Proceedings of the 7th siam international conference on data mining*, pages 629–634. Citeseer, 2007.