Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Bachelor's Program in Computer Science

Bachelor's Thesis

Predicting Security Vulnerabilities from Function Calls

submitted by

Christian Holler

on September 26, 2007

Supervisor:	Prof. Andreas Zeller
Advisor:	Dipl. Inform. Stephan Neuhaus
Reviewers:	Prof. Andreas Zeller Prof. Wolfgang J. Paul

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbruecken, September 26, 2007

Christian Holler

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbruecken, September 26, 2007

Christian Holler

Contents

Chapter 1

Introduction

Security vulnerabilities in software do not only pose a risk to our data and privacy but also cause major financial loss[1]. Unfortunately, such vulnerabilities are hard to find, especially in huge projects, and searching for them is not only a time-consuming but also an expensive process. Of course, one could completely avoid security vulnerabilities by completely specifying the software and then proving the correctness, however, this is rarely done nowadays, mainly because the complexity makes the whole process very expensive and time consuming.
In this thesis, we will see a statistical approach to aid in this search. More precisely, I will evaluate the suitability of *functions calls* as statistical predictors for security vulnerabilities, using the Mozilla Project [2] source code and a provided database of known vulnerabilities. The main reason for the choice of function calls as a predictor is that the Vulture project [1] has already reported good results with *imports* as a predictor. The required database was previously mined from the Mozilla CVS and contains all known security vulnerabilities from 2005 to January 2007. Figure 1.1 explains the relationship between all involved elements, both the vulnerability database and the source code serve as an input for our machine learning process which again creates a predictor that can be used for several purposes. In this case, a new component is tested to find out how likely it is to contain vulnerabilities. More information on Mozilla, the mining process and vulnerabilities can be found in Section 3.1.
The first question that arises when we look at vulnerabilities and function calls is most likely

"Are there correlations between function calls and security vulnerabilities?"

If we find such a correlation, we need to evaluate if it is suitable to predict vulnerabilities. This step is described in Section 3.3. If the evaluation shows an acceptable accuracy, this correlation could be used to narrow down the search space for unknown vulnerabilities in large projects, effectively saving time and money. Another question consequently is

"Can these correlations be used to accurately predict security vulnerabilities?"

If this is the case, this method could also be used for several other practical applications. For example developers could be alerted when they introduce new code that

[1]An FBI study published in 2005 states a loss of 67 billion USD through computer crime

contains possibly risky patterns of function calls, preventing security vulnerabilities from reaching production code. So we ask ourselves

"How well does this method work in practice?"

Both to answer this question and to give an example for predicting, I will compile a list of newly found vulnerabilities in the period from January to July 2007 and evaluate, how well these vulnerabilities can be predicted using a "leave one out"-method to do predictions for single components (See Section 3.4).

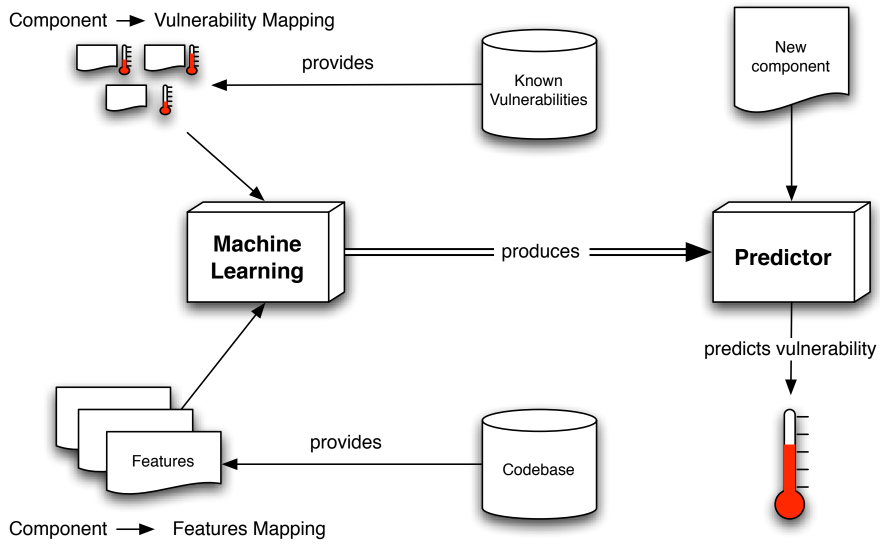


Figure 1.1: Relationship between CVS, vulnerabilities and predictions

Chapter 2

Related Work

As already mentioned in the introduction, this thesis is based directly on the Vulture project [1], developed by Stephan Neuhaus et al., where they evaluate the quality of imports as a predictor in a similar way. Because their evaluation showed good results and function calls provide more detailed information as compared to imports, the results for function calls are expected to be even better.
The idea of looking at the history of components was already applied in a work by Schröter et al. [19] who investigated correlations between imports and *general defects*. However, this work was not focusing on security vulnerabilities.

Further related studies are:

The evolution of defect numbers, researched by both Ozment et al. [11] and Li et al. [10]. These studies deal with the evolution of the numbers of security vulnerabilities and defects over the time and came to different conclusions. Ozment et al. report a decrease in the rate at which new vulnerabilities are reported, while Li et al. report an increase. However, neither allow a mapping between components and vulnerabilities or a prediction.

Describing security vulnerabilities with models, researched by Chen et al. [15]. In their paper they deal with the difficult task to depict and reason about security vulnerabilities using data from bugtraq and source code examination. This work and mine have in common that both a vulnerability database and the source code are used as starting point. Chen et al. use finite state machines to model vulnerabilities and using this model, many common types of vulnerabilities are decomposed to simple predicates to understand how and why they occur, however, they do not predict new vulnerabilities.

More practical approaches to reduce the number of security vulnerabilities, include:

Testing the binary. These approaches are directed at the running program and comprehend methods like fuzz testing [16, 17] and fault injection [14]. Briefly, these methods find vulnerabilities by provoking faults with either specially synthesized or random input and monitoring the program behavior.

Static source code analysis, focusing on *specific types/forms of security vulnerabilities*. Specifically, there are several tools for the detection of buffer overflows in C and C++ such as "Mjolnir", described by M. Weber et al. in their work [12].

There are also more generic scanners, such as ITS4, a security vulnerability scanner for C and C++ code developed by Viega et al. [13]. This scanner is based on known risky patterns that are often associated with security vulnerabilities. However, most of these methods are either specific to one or several types of vulnerabilities, or require static patterns (used in ITS4), that need to be updated constantly. Also, such methods might not able to recognize vulnerabilities which have a bigger context, across several components.

Runtime detection and prevention of exploits. There are implementations for runtime countermeasures available for the Linux kernel, such as PaX which is a part of the grsecurity package [7]. PaX generally aims at preventing any form of arbitrary code execution, e.g. through buffer overflows or heap corruption. Grsecurity also implements other methods to lower the risks of a successful attack, such as role-based access control lists. Another familiar implementation of role-based ACLs is SELinux [9].

However, these methods were not designed to find and prevent security vulnerabilities, but to lower chance for a successful exploit and its impact once an exploit was successful.

Chapter 3

Method and Implementation

3.1 General Definitions and Information

As stated before, the Mozilla project plays an important role because it is used for all evaluations described in this thesis. We should therefore have a look at relevant data from this project, especially at the process of discovering/fixing vulnerabilities and how many vulnerabilities have been discovered in the past.
Mozilla is a fairly large project, it currently consists of 10452 C/C++ components (see Definition 3.1) and lots of additional javascript files. The source code has a total size of about 800 megabytes. Since 2005, the project of course had several security problems, and in total, 424 components were affected by security vulnerabilities. Compared to the total number of components, this is a *very low* percentage (approximately 4%).
The process of reporting, publishing and fixing vulnerabilities has been standardized in the Mozilla project. Vulnerabilities are reported using Bugzilla[1], a bug management system developed by the Mozilla group. Reports are generally filed by the person that discovered the vulnerability; most times these are Mozilla developers or external people such as security specialists and other developers. However, sometimes vulnerabilities are discovered because they are already being exploited in the wild (*0-day exploit*).
Each report has a unique *bugid* and every CVS commit that is associated with this bug includes the bugid for later reference. This is an important fact for definition 3.3. Once a bug has been confirmed and fixed, the report is closed. If the bug was security related, a *Mozilla Foundation Security Advisory (MFSA)* is published[2], as soon as fixed version of the product is released.

3.1.1 Definitions

Some of the terms used in this thesis have no universally valid definition. In order to aid reproduction of my results, we therefore define some core terms.

Definition 3.1 (Component). Unlike in other languages such as Java, declarations and definitions in C/C++ often reside in different files. Declarations are made in header files

[1] http://www.bugzilla.org/
[2] See http://www.mozilla.org/projects/security/known-vulnerabilities.html for a list

whereas definitions are made in the corresponding source file[3]. Nevertheless, these files form an unseparable entity that we consider as a component.
This model is suitable because a security vulnerability that affects one of these files of course affects the whole component. It makes no sense to assign a vulnerability only to a source file, for example, and not to its corresponding header file.

Definition 3.2 (Component Naming Convention). To give components unique names, each component name starts with its path relative to the toplevel Mozilla directory. In many cases, there is a header and a source file as described previously. In this case, the file extensions are omitted for component naming. Sometimes though, a component consists of a single file, in this case, the component name will contain the full filename including the file extension. The following examples clarify this:

```
mozilla # ls xpinstall/src/nsInstallResources*
xpinstall/src/nsInstallResources.cpp
xpinstall/src/nsInstallResources.h
```

$\longrightarrow$ Component name: xpinstall/src/nsInstallResources

```
mozilla # ls js/src/jsbit*
js/src/jsbit.h
```

$\longrightarrow$ Component name: js/src/jsbit.h

Of course, to reference to a specific component in this document the path may be omitted where this does not cause any ambiguities.

Definition 3.3 (Vulnerable Component). To associate components with vulnerabilities, we will obtain the bugids for vulnerabilities from the Mozilla security advisory website and then use the Mozilla CVS to search for changes that contain this bugid, as shown in Figure 3.1. As explained earlier in this chapter, all changes that are related to a given bugreport include the bug id in the commit message.
Therefore we consider every component as vulnerable that has a change log containing a bugid that is associated with a vulnerability. However, this definition is very broad and has some problems, see Section 5 for possible problems with this definition.

Definition 3.4 (Neutral Component). A component is neutral when it is not vulnerable according to definition 3.3. The word “neutral” especially implies that we do not know exactly whether this component actually has an undiscovered vulnerability. Therefore, “neutral” should never be mixed up with “invulnerable” as there might be vulnerabilities present that nobody is aware of.

3.1.2 Environment

All programs that were written in the course of this thesis were in Perl [3] and R [4]. Perl is used for parsing, preprocessing of input data and postprocessing of results, whereas R is used for the actual predictions. Especially for our machine learning parts, the e1071 [6] package for R provides the required algorithms. Also, e1071 is one of the few packages that support sparse matrices for several algorithms during the whole process of computing and predicting. As we will see later, this provides a great speedup

[3]This is an idealized model, in practice, short definitions are often placed in header files directly and declarations can also be made in source files.

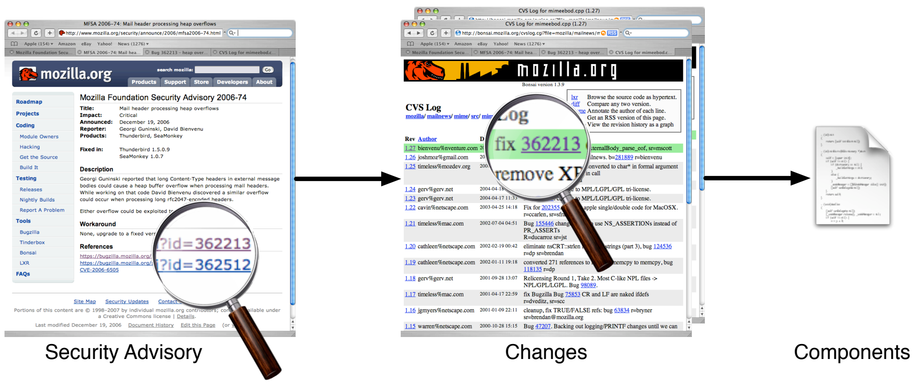


Figure 3.1: Mapping vulnerabilities to components through advisories and bug reports

for predictions. Although all programs were written for Linux, they could also run under Windows with slight modifications since both Perl and R exist for Windows as well.

3.2 Feature Extraction from Source Code

3.2.1 Function Calls

The Mozilla project is mainly written in C and C++, hence these languages are our target languages. In these two languages, a *function call* is an expression that could cause the control flow to be transferred to a function when it is executed.[4] A function call is characterized by the name of the function and a parenthesized list of arguments.

Another characteristic which is specific to C and C++ is the frequent use of preprocessor statements to modify the code in the preprocessing stage of the compilation process, to satisfy specific library or operating system requirements. Therefore, most of this source code cannot be preprocessed without tieing oneself to a specific platform such as Windows or Linux, effectively causing a loss of source code for other platforms. Also, preprocessing sometimes requires external headers to be present, especially for historical versions these are old headers that we cannot supply.

As a consequence, extracting the function calls from the unpreprocessed source code is the preferred way. However, dynamic dispatch, i.e. parsing with type information would require compilable source code and even a full static parsing is blighted by syntax errors caused by some preprocessor statements; see Figure 3.2.

An acceptable solution is to simply treat all occurrences of *identifier*`(...)` and *identifier*`<...>(...)` as function calls. Of course, these patterns do not match only function calls. There are other constructs, both desired and undesired ones, which will be matched as well. For example:

[4]This cautious phrasing is necessary because of the possibility of inlining.

Function definitions

```
JS_PUBLIC_API(int64) JS_Now() { return PRMJ_Now(); }
```

Extern Forward declarations

```
extern JS_PUBLIC_API(int64) JS_Now();
```

Macro calls

```
MOZ_COUNT_CTOR(nsInstallLogComment);
```

Constructor calls

```
ie = new nsInstallFile( this, [...] );
```

Some keywords

```
if (buffer == nsnull || !mInstall) return 0;
```

Initialization lists

```
nsXPItem::nsXPItem(const PRUnichar* aName) : mName(aName)
```

C++ functional-style casts

```
int(curWidget->fieldlen.length)
```

Some of these false positives can be avoided easily. Keywords for example are easy to exclude using a static list of keywords[5] and function definitions can be recognized by the opening curly bracket following the pattern.
Excluding these two points, we still have several sources of false positives, some of which are desired and some not. Both macro and constructor calls behave similar to our definition of function calls and should therefore be treated in the same manner. However, forward declarations, initialization lists and type casts are rather undesired. Even though I haven't found any easy heuristics which can be statically applied to exclude these from our matching, they don't seem to spoil the results as we will see later in Chapter 4.

[5]The full list of keywords is available at http://www.cppreference.com/keywords/index.html

```
#ifdef XP_OS2
  if (DosCreatePipe(&pipefd[0], &pipefd[1], 4096) != 0) {
#else
  if (pipe(pipefd) == -1) {
#endif
    fprintf(stderr, "cannot create pipe: %d\n", errno);
    exit(1);
  }
```

Figure 3.2: Extract from `nsprpub/pr/tests/sigpipe.c`, lines 85ff. Parsing C and C++ is generally only possible after preprocessing: attempting to parse these lines without preprocessing results in a syntax error.

3.2.2 Implementation

The final parser includes all methods to identify function calls as described above as well as common parsing methods such as exclusion of comments, strings and preprocessor statements from our search space. It takes up only 9 kilobytes or 346 lines of code.
The parser takes all files, that belong to the component to analyze, as parameters. After extracting the function calls from those files, it determines the component name according to Section 3.2 and outputs the results in the following format:

```
<component name>: functionA/count functionB/count [...]
```

Example:

```
js/src/jsstr: AddCharsToURI/6 js_ExecuteRegExp/4 [...]
```

Because the parser outputs its results for a single component, a second tool is required that invokes the parser with the respective files. This tool is called the **project parser** and goes through the project directory structure, searching for relevant files, grouping them together into components and then invokes the parser for each file group.
For more information about the parser speed on the whole project, see Section 3.5.

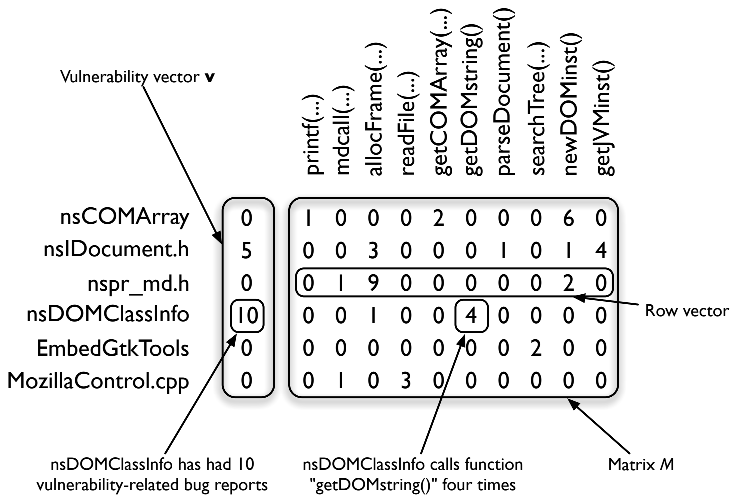
3.3 Automated Testing

The data provided by the parser can be interpreted as a matrix M which represents the relationships between all functions f_j in the project and its components c_i. The matrix entries describe the number of occurrences of a function within a component, i.e. m_{ij} is the number of occurrences of the function f_j in the component c_i.
Using the provided data about past vulnerabilities in the project, we can also associate each component c_i with a value v_i indicating the number of past vulnerabilities in this component, yielding a vector $\mathbf{v}$ called the vulnerability vector.
Figure 3.3 shows an example for M and $\mathbf{v}$ with fictitious values. Both M and $\mathbf{v}$ together form the basis for all further experiments and their evaluation.
To find statistical correlations between function calls and vulnerabilities in this dataset, we will use *machine learning*. When using machine learning, generally a model f is created using a suitable *training set*. The general recommendation for the size of this training set is $2/3$ of the whole dataset and we will follow this recommendation for our evaluation setup. The general idea is to split up the dataset into two parts,

the training set which consists of the matrix M_T and the vector $\mathbf{x}$, the parts of $\mathbf{v}$ that correspond to M_T, and

the validation set which consists of the matrix M_V and the vector $\mathbf{y}$, the parts of $\mathbf{v}$ that correspond to M_V,

and then use the training set to generate a model f that we can apply to the validation set.
In our case, we will use *Support Vector Machines* [5] as a special case for machine learning, because SVMs can also handle data that is not linearly separable. Also, they are less prone to overfitting than other machine-learning methods, such as k-nearest-neighbors[20].



	v	printf(...)	mdcall(...)	allocFrame(...)	readFile(...)	getCOMArray(...)	getDOMstring()	parseDocument()	searchTree(...)	newDOMinst()	getJVMinst()
nsCOMArray	0	1	0	0	0	2	0	0	0	6	0
nsIDocument.h	5	0	0	3	0	0	0	1	0	1	4
nspr_md.h	0	0	1	9	0	0	0	0	0	2	0
nsDOMClassInfo	10	0	0	1	0	0	4	0	0	0	0
EmbedGtkTools	0	0	0	0	0	0	0	0	2	0	0
MozillaControl.cpp	0	0	1	0	3	0	0	0	0	0	0

Figure 3.3: Example for feature matrix M and vulnerability vector $\mathbf{v}$.

However, because only approximately 4% of all components are vulnerable, randomly selecting 2/3 from all components might yield a training set which contains none or to few vulnerable components. This biased distribution of vulnerable components would lead to bad prediction results because the SVM would not have enough examples for vulnerabilities. To avoid this, we will use stratified sampling. With this method, the dataset is first split up into vulnerable and neutral components and then a random split is done for each of these sets. After doing so, the 2/3 vulnerable components and the 2/3 neutral components are merged into a single training set, the same is done for the testing set. Figure 3.5 illustrates the process.
However SVMs support different methods of operation, classification and regression, and the concrete evaluation of the prediction results depends on the operation mode. We will now have a look at these modes and how to evaluate them.

3.3.1 Classification

In classification mode, the vulnerability vector needs to be binarized to $\mathbf{v}'$ with $v'_j = 1$ if $v_j > 0$ and the SVM prediction for a component is either 0 (not vulnerable) or 1 (vulnerable). After training the SVM with the training set, that is M_T and $\mathbf{x}$, we can apply it to our validation set, yielding a result vector $\hat{\mathbf{y}}$.
To assess the quality of such predictions, there are two generally accepted measures. First, there is the *precision*, which describes the correctness of our positive predictions, i.e. how often components are actually vulnerable compared to all components predicted to be vulnerable. Then, there is the *recall*, which describes how many vulnerabilities were predicted at all, compared to those that *could* have been found. To calculate these values, we need to compare the predicted result vector $\hat{\mathbf{y}}$ to our known result vector $\mathbf{y}$. When comparing these vectors, there are four different situations for each index j:

True positive: Both y_j and $\hat{y}_j$ are 1.

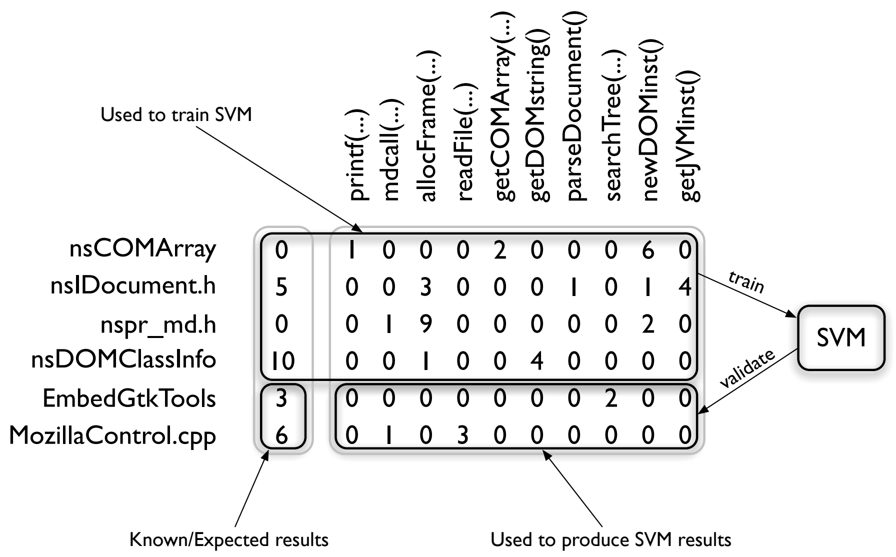


Figure 3.4: General evaluation strategy.

False positive: The actual value y_j is 0 but the predicted value $\hat{y}_j$ is 1.

True negative: Both y_j and $\hat{y}_j$ are 0.

False negative: The actual value y_j is 1 but the predicted value $\hat{y}_j$ is 0.

Once we have counted how often each situation happens, these values can be used to compute the precision and recall values which represent the accuracy and the completeness of the prediction as follows:

$$\text{Precision} = TP/(TP + FP) \tag{3.1}$$

$$\text{Recall} = TP/(TP + FN) \tag{3.2}$$

A high precision means a low false positive rate and a high recall means a low false negative rate. Of course, a high value for precision or recall alone is unsuitable because a high precision and a very low recall means that we hardly make mistakes in our positive predictions but we only find a small count of vulnerabilities compared to the total number. On the other side, a high recall with a low precision means that we identify lots of vulnerabilities compared to how many we could find, but we also produce so many false positives that the final result is useless because we can't distinguish between them and the true positives. Consequently both values needs to be in balance and preferrably high.

3.3.2 Regression

In regression mode, the SVM trains with the unmodified vulnerability vector, using the absolute values to predict the number of vulnerabilities per component. These values don't need to be accurate, but can be used to rank components.

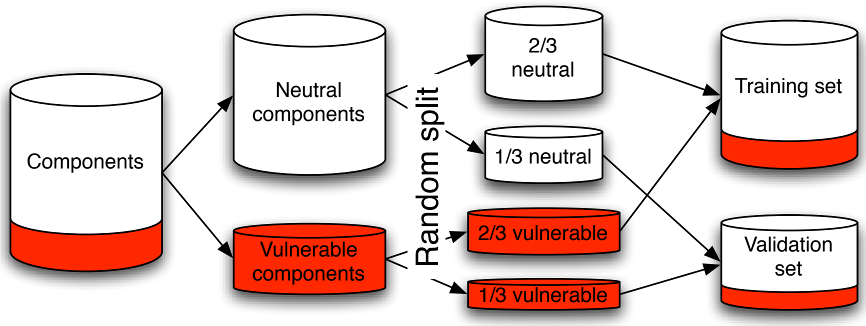


Figure 3.5: Stratified sampling.

		Actually has vulnerability reports		
		yes	no	
Predicted to have vulnerability reports	yes	True Positive (TP)	False Positive (FP)	Precision
	no	False Negative (FN)	True Negative (TN)	
		Recall		

Figure 3.6: Precision ($TP/(TP+FP)$) and recall ($TP/(TP+FN)$).

Being able to produce a ranking of the most vulnerable components, especially for those that do only have few or even no known vulnerabilities, could help developers focus their efforts on a smaller group of components, effectively maximizing their success in finding vulnerabilities whilst minimizing effort.

The reviewers of the Vulture paper [1] rejected the use of the *Spearman Rank Correlation Coefficient* [18] to evaluate how exact the predicted ranking reflects the actual ranking. Instead, they suggested a cost model that matches the practical scenario where a manager has to distribute limited resources to audit components:

Assuming that we have the resources to test T components, then these would be best spent on the top T most vulnerable components, however the order inbetween these top T is irrelevant. Ideally, the predicted top T components would be the same as the actual top T components, no matter their order. To evaluate the quality of our prediction, we hence could compare how many vulnerabilities we can fix with our prediction compared to the ideal case:

With m being the number of components and $\hat{\mathbf{y}}$ being the predicted vulnerability vector, we define $p = (p_1, ..., p_m)$ as a permutation of $1, ..., m$ such that $\hat{y}_p = (\hat{y}_{p_1}, ..., \hat{y}_{p_m})$ is sorted in descending order. Fixing component p_j fixes y_{p_j} vulnerabilities, hence fixing the top T predicted components fixes $F = \sum_{1 \le j \le T} y_{p_j}$ vulnerabilities. However, with the optimal ordering, with q and y_q defined accordingly to p and $\hat{y}_p$, we could have fixed $F_{\text{opt}} = \sum_{1 \leq j \leq T} y_{q_j}$ vulnerabilities. We can now use the quotient $Q = F/F_{\text{opt}}$ to indicate the quality of our prediction. Because $0 \leq F \leq F_{\text{opt}}$, Q is a value between 0 and 1 and higher values indicate a better ranking.

In a typical situation, the total number of vulnerable components is much smaller than the number of neutral components, therefore a random selection will almost always give $Q = 0$ for small T. In order to be useful in practice, we hence expect a value for Q that is considerably greater than zero. A good threshold would be 0.5 which means that at least half of the effort was spent meaningfully.

3.3.3 Implementation

The implementation for the automated testing environment consists of several parts:

The preprocessor translates the parser output described in Section 3.2.2 into a *sparse matrix format* that is readable by the `read.matrix.csr` function provided by the e1071 package. This format is highly suitable because the matrix is only sparsely populated and writing out the whole matrix would consume several gigabytes of space whereas the sparse matrix only takes up 2 megabytes. During the translation, the matrix is also sorted in both dimensions using the lexical order of the function names and components as the sorting criteria. Additionally, filters can be applied like binarization for the data values, i.e. cutting all non-zero values to 1 and several filter criterias which remove functions from the matrix. One motivation for the binarization filter is that absolute values also depend on the size of a component, so a correlation between the size and the number of vulnerabilities could degrade the results. Of course, the preprocessor is also able to output the vulnerability vector, a index-to-component mapping and several other useful pieces of information.

The R scripts are responsible for splitting our data as well as training the SVM and using it to predict the vulnerabilities for our validation set. After reading the sparse matrix and the vulnerability vector, a stratified sampling is done and the training process is started with the training set. Once the SVM is trained, the prediction is done for the testing set and results are written to the terminal. For classification, this is simply the table containing the true/false positives/negatives, but for regression, the scripts already outputs the results as described in Section 3.3.2. Additionally, each script outputs its random split indices into a file to allow reproduction of our results.

The simulator supervises the execution of the R scripts. To rule out that the results are only good for one random split, the simulator supports repeating the same script for a given amount of time, each time using a different random split and all results are collectively written to one file. Even though a single prediction process is quite fast, the simulator supports threading for an additional speed gain on multiprocessor systems.

Several evaluation scripts are then used to summarize a result file for a simulation instance, e.g. calculating the median of result values; see Figure 3.7

3.4 On-Road Test

In addition to the automated tests described above, an on-road test with the Mozilla project and current vulnerabilities would demonstrate that this is indeed a method for *practical use*. Therefore I decided to do such a test and the setup is described below.

3.4.1 Test Setup

The automated tests used in this thesis only have information about vulnerabilities discovered up to January, 2007; however, at the time of test, it was already July and lots of new vulnerabilities were discovered. A sensible step would be to use these new vulnerabilities to test the predictive power of our method. In the first step, we hence need to find all vulnerabilities and map these to components according to Section 3.3. The second step is prediction and evaluation. The prediction will work using a "leave one out"-method, in detail a single prediction will be done as follows:
Assuming that we want a prediction for component j $(1 \leq j \leq m)$, we use *all* components $(1, \ldots, m)$, *except* the component j itself, to train the SVM. After doing so, the SVM is used to predict the component j. These steps will be done in regression mode because regression allows us to better extract a small number of components that are very likely to be vulnerable. Figure 3.8 illustrates a prediction step for a single component.
In our test, the above steps will be done for all components 1, ..., m and the result for each component is recorded. Of course we will use the *old* vulnerability vector that does not contain the new vulnerabilities to do our predictions. After the whole test is finished, we will look at the predicted top 20 of components that were previously flagged as neutral and see how many of them are now affected by a new vulnerability and how the vulnerability affects the component[6].

3.4.2 Implementation

Prior to any implementation, we first have to get information about new vulnerabilities. We can extract these from *http://www.mozilla.org/projects/security/known-vulnerabilities.html*. All Mozilla security advisories are listed there with there respective *bugzilla* bugid. For our purpose, I compiled a list of new vulnerabilities by hand in a simple parseable format, listing the MFSA and all bugids related to this MFSA.
Of course, this information could also be parsed from the Mozilla website automatically[7], but in this case, it was easier to compile the list by hand. In addition to this list, we need the full Mozilla CVS which takes about 3.6 gigabytes of downloading. With these requirements satisfied, the automated scripts can do the rest. They again consist of several parts:

The CVS parser gets the vulnerability list as input and searches through the CVS log of each file looking for bugids that are mentioned in this list. It outputs all components that are affected by any of these bugids with the MFSA number and the bugid. If a component is affected by two MFSAs and/or bugids, the parser outputs the component twice with the respective information. This is essentially a simple re-implementation of the algorithm outlined in the Vulture paper.

[6]Inspecting how a vulnerability affects a component of course does not work automatically but requires a qualified human.

[7]The Vulture project includes a parser for this task.

The parser postprocessor compares the components found by the CVS Parser to our list of components that are already vulnerable. For our test, we only need the components that are still listed as neutral in our vulnerability vector. Additionally, this tool outputs more useful information, e.g how many components are "repeaters"[8].

An R script reads the vulnerability matrix and vector and then starts the "leave one out" prediction, outputting the results for each component into a single file.

An evaluation script reads these results, calculates the top 20 of components that were previously listed a neutral and are now predicted to be vulnerable, with the predicted number of vulnerabilities as sorting criteria. Then it compares, how many of these components are actually vulnerable now according to our new vulnerability data.

3.5 Performance

Even with the best results, a method is still not suitable for practice when it takes months or even years to produce results. To demonstrate that the methods described here also satisfy common performance criteria, table 3.1 shows a detailed performance breakdown for all time intensive processes.

Process	Time
Function call extraction	8 m
SVM Training w/classification and regression	0.5 m
CVS Parsing for on-road test	2.5 m
SVM "leave one out" regression	6 d

Table 3.1: Approximate running times for major method parts described in this thesis

Because the actual prediction parts require under a minute to run, they could even be used live, while the programmer is still writing the code, or when the code is checked in, providing full flexibility for the deployment of methods based on these predictions. One of the reasons why these parts are so fast is the use of *sparse matrices*. With full matrices, the training and classification/regression would take almost 20 minutes.
Please note that the implementation for the "leave one out" regression is not threaded. Using threads and running this on nowadays commonly used multi processor machines would lead to a massive speed gain as the method is schedulable in parallel on component level.

[8]Components that were vulnerable before and now have even more vulnerabilities

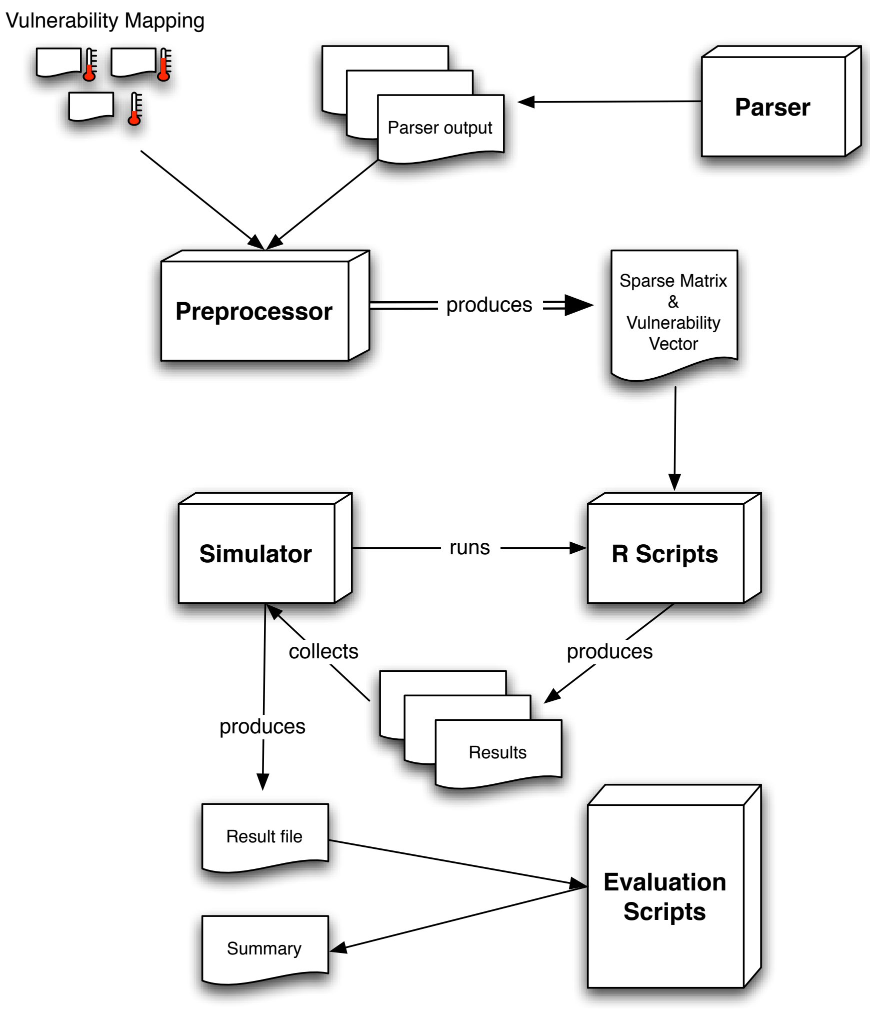


Figure 3.7: Interaction between implementation parts

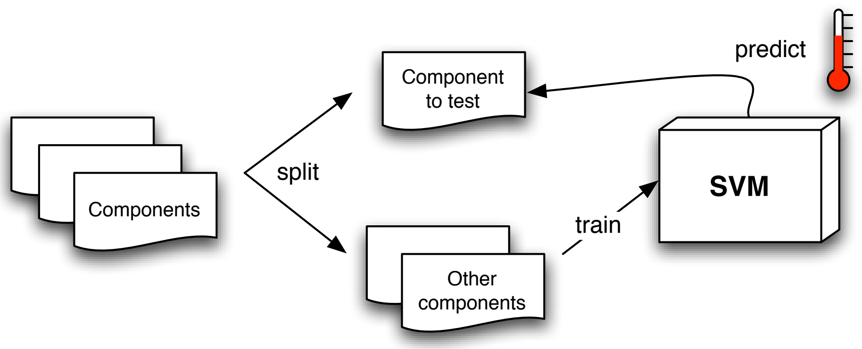


Figure 3.8: Single step in “leave one out” prediction

```
# MFSA  bugid [bugid ..]
2007-25 369211 370127
2007-24 387333
2007-23 384384
```

Figure 3.9: New vulnerabilities since January 2007 in simple format, hand compiled.

Chapter 4

Evaluation of Results

4.1 Automated Testing Results

In this section we will have a look at the results for our automated tests described in Section 3.3. However, we will not only look at the best results, but also at other settings for our matrix preprocessor and see how they changed the results. These are:

- The "binary" setting, each matrix entry m_{ij} is replaced with 1 if $m_{ij} > 0$ and else with 0.

- The "stripped" setting, functions which only appear in one component are ignored, as well as function names with a length of 2 or less. The aim is to remove irrelevant functions to improve the results. When using this setting, the matrix width is reduced from 93265 to 40224, however, the execution time is hardly affected.

Both for classification and regression however, it turned out that using the "binary" setting leads to better results, even though binary values provide less precise information.

4.1.1 Classification

For classification, we measure precision and recall for our evaluation as described in Section 3.3.1. Table 4.1 shows the median results for several settings, based on 100 splits for each setting.

Dataset	Precision	Recall
Original	50.77%	40.41%
Binary	**69.95%**	**40.14%**
Stripped	49.08%	40.94%
Binary and stripped	64.61%	43.71%

Table 4.1: Median values according to cost model described in Section 3.3.2 across different datasets

The results do not only show that binary values lead to a much better result, but also that the SVM is able to handle unimportant data that is removed by our filters, the results are almost the same. Figure 4.1 shows the detailed distribution of all 100 pairs for the binary dataset. But even more important, for the best dataset

**70% of all predictions are correct,
and 40% of all known vulnerabilities were found.**

When we compare this to a random selection, with 10452 components and only 424 vulnerable (approx. 4%), then we see that these values provide a tremendous advantage to simple random guessing.

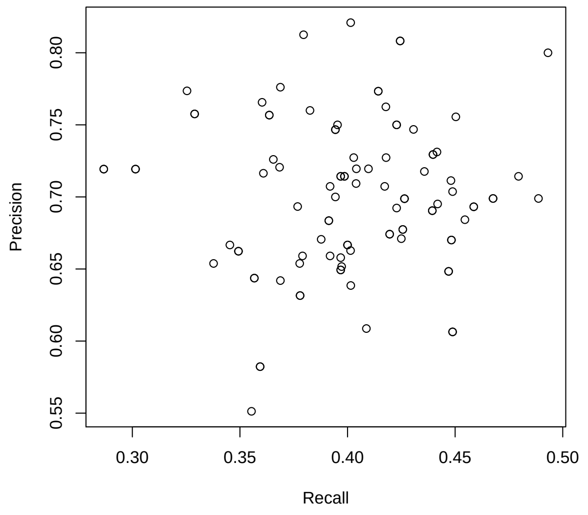


Figure 4.1: Scatterplot for 100 precision/recall pairs

4.1.2 Regression

Dataset	Top 10	Top 30
Original	73.33%	72.19%
Binary	80.91%	84.81%
Stripped	75%	68.28%
Binary and stripped	**82.47%**	**83.33%**

Table 4.2: Median values for top 10 and top 30 across different datasets

As expected, the distribution of results across the different datasets is similar to the results in classification and the binary datasets gives the best results. However, in this case we see that the stripped binary dataset is slightly better than the binary set, however these are only minor differences. With the stripped binary set

82% of all possible top 10 vulnerabilities and 83% of all possible top 30 vulnerabilities were found.

As already explained in Section 3.3.2, with the ratio of vulnerable components to neutral components being very low (roughly 1:25), a random selection would almost always lead to a total of 0 vulnerabilities found.

4.2 On-Road Test Results

Since January 2007, 68 components that were neutral needed to be fixed because of vulnerabilities by July 2007. After examining the predicted top 20 of all 10028 neutral components, a total of 6 components were now actually affected by one or more new vulnerabilities. That means with only 0.678% of components that are now vulnerable, we found 8.8% only by looking at the Top 20, i.e. more than *twelve times* the amount of vulnerabilities compared to a random selection. After inspecting the bug reports and CVS commit messages of these vulnerabilities, it was clear that all except one of these components were either the source of the vulnerability or directly involved with the vulnerable component. The remaining component that was predicted to be at rank 1, turned out to be a false positive. A version string inside the component was changed with a vulnerability bugid in the commit message, however, the component had no direct relation to the vulnerability. Figure 4.3 shows the list of relevant components and their result of manual inspection.

#	Component Name	Bug Id	Detailed information
2	`jsscope`	Bug 381374	Assertion failure in component, allowing to crash program
		Bug 364657	Possible crash in another component, fix includes major code addition to this component
7	`nsSliderFrame`	Bug 344228	Component was using the `nsIScrollbarMediator` component in an insecure way
9	`nsTableRowFrame`	Bug 370360	Actual bug in `nsTableRowGroupFrame` but components seem very similar in their domain
10	`nsFrameManager`	Bug 343293	Crash bug, missing assertions and checks in this component
	`nsFrameManager`	Bug 363813	Component not directly affected but source code changes made because of this bug
18	`nsIContent.h`	Bug 382754	Bug report not publically accessible but CVS shows that this is one of two files changed with identical commit message

Table 4.3: Components in our prediction that are now actually vulnerable, with # indicating the predicted rank

With 4 hits in the predicted top 10, based on the vulnerabilities discovered in only half a year, this is a really good result. If the Mozilla team had received this information in January, they could have found these vulnerabilities even faster by concentrating their efforts on the top 10 of our prediction. Altogether, the on-road test showed that this method is indeed providing excellent results, suitable for practical application.

Chapter 5

Threats to Validity

There are several possible threats to the validity of my work, those are:

Limited target codebase used for evaluation. The whole process of predicting and evaluating was done with a single project, the Mozilla Project. Hence, the observed correlations could be specific to this project only. However, it seems unlikely that the Mozilla Project is such a special project compared to other major codebases, so this is rather a minor threat.

Defective tools. The tools used in this thesis might contain bugs themselves that lead to falsified results. This is not unlikely, I personally found 2 bugs in the sparse matrix implementation of e1071. However, the Vulture project was using full matrices in an earlier stage and we verified the Vulture results using the sparse implementation and the chance that both the sparse and the full implementation suffer from the same bug is very low.

Inaccurate feature extraction. In Section 3.2 we already saw that it is not perfectly possible to statically extract function calls from C/C++ source code. The inaccuracy in this process could lead to falsified input data, however the results don't seem to be spoiled by this.

Broad definition of "vulnerable". In general, we call a component *vulnerable* when it is directly affected by a security vulnerability, however when doing automated mining of these information, it is often not decidable whether a component is actually affected by a vulnerability in the context of security. When a vulnerability is fixed, other components require changes sometimes, because a component or function interface changed, without however being directly affected by the security impact of the vulnerability. This would lead both to inaccurate input (in the vulnerability mapping) and inaccurate comparative data (e.g. in the on-road test). A good example here is a component that was on rank 1 of our prediction in the on-road test, but the change that marked this component as vulnerable was only a version number change. The impact of this threat has yet to be evaluated but is not in the scope of this thesis.

Evaluation misses chronological context. A prediction always implies a chronological order, the prediction generally depends on past data and the components that are analyzed lie in the present. The automated evaluation however, does not honor this context when doing the splits for training and validation set. This could lead to better results than one would get when forcing a chronological order of components. The impact of this threat has yet to be evaluated but is not in the scope of this thesis.

Evaluation misses grouping of vulnerable components. As described earlier in this section, a vulnerability often causes multiple files to be fixed at once, be it because of interface changes or because a bug spans across several components. All changed components are marked as vulnerable according to our definition in Section 3.3. See Figure 5.1 for a simple example.

However, during the automated evaluation, these groupings are broken up and each vulnerable component is treated as if it was affected by an its own bug. Hence it is likely to happen that components that are affected by the same vulnerability are separated, some go into the training set and some into the validation set.

If the correlations between components that suffer of a single vulnerability are stronger, than those between components that suffer from different vulnerabilities, then this might falsify the results. Figure 5.2 shows such a situation, where basically a vulnerability "predicts itself".

The typical scenario for a prediction of new vulnerabilities though is a "past to present" relationship between the known and the yet unknown vulnerabilities. The situation described in Figure 5.2 and hence the whole automated evaluation process, is therefore very unlikely in this application case and therefore, the results in a practical prediction scenario could be worse than those observed in the evaluation. Figure 5.3 shows this in a simplified way.

However, it is not given that the correlations inside such a group are stronger than the correlations between different vulnerabilities, and even if this should be case, it is not possible to say to what extend the results are affected by this. Researching these effects is not in the scope of this work.

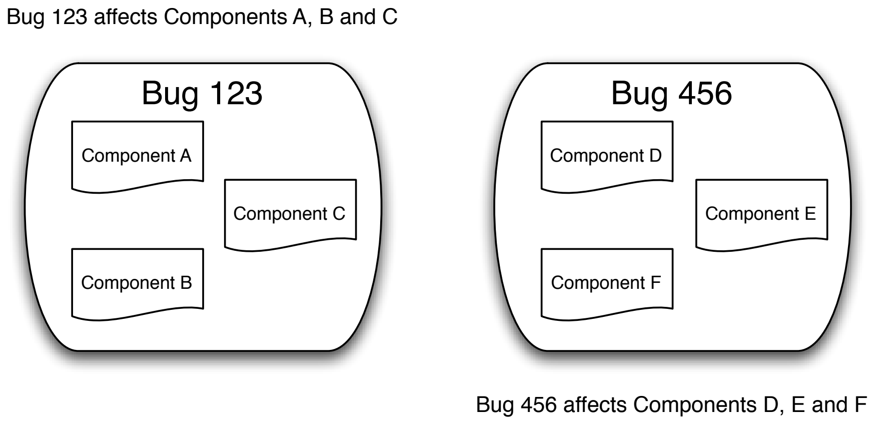


Figure 5.1: Initial situation, two vulnerabilities, each affects several components.

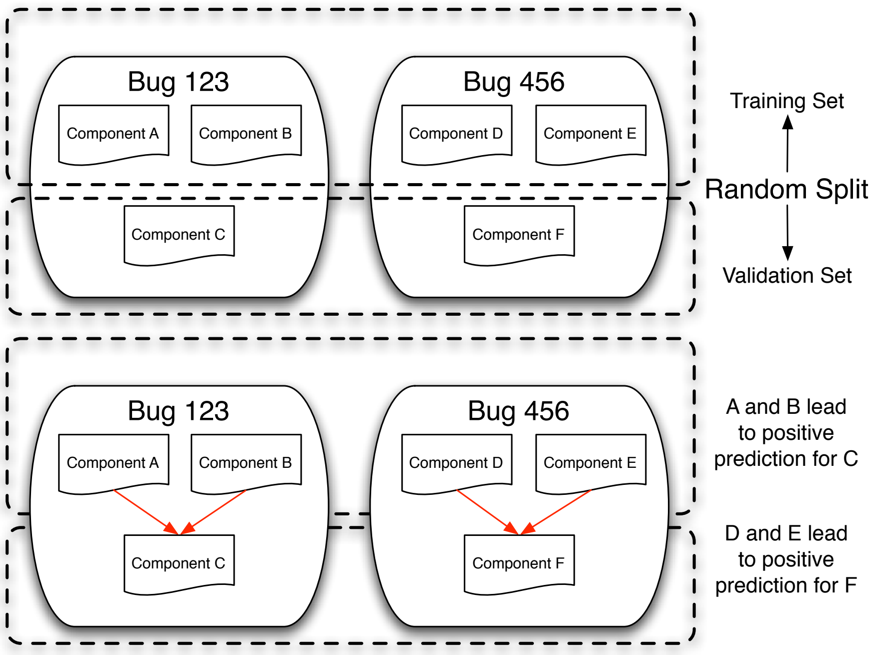


Figure 5.2: The random split goes across component groups.

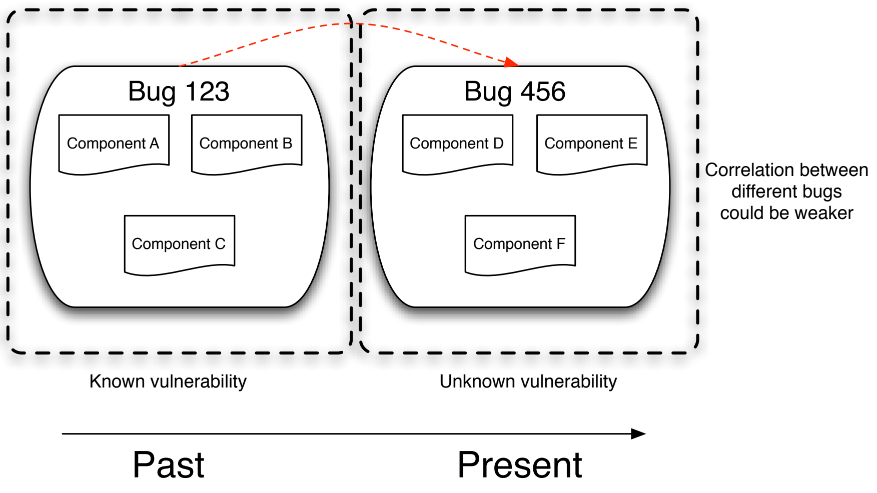


Figure 5.3: Correlation between different vulnerabilities could be weaker, making the method less effective in a typical prediction scenario.

Chapter 6

Conclusion and Future Work

The automated evaluation tests clearly showed that there is a correlation between function calls and security vulnerabilities that can be used to predict unknown vulnerabilities. We also saw that these predictions are not only superior to random selection under the aspects of a cost-benefit analysis, but also highly precise (70% precision) in classification. According to our cost model for regression, our predictions also provide results that can be used to greatly increase the efficiency of finding security vulnerabilities.
Also, with the on-road test we saw a method for practical application in large projects, showing us that this is not a purely theoretic result.
With this success, we can resolve on several future objectives:

Evaluating the method with different projects. Gathering further results with other projects eleminates the threat of limitation to a single project, strengthening the position of the described methods amongst other scientific approaches.

Evaluating other features. The framework described and used in this thesis is not limited to function calls as features. Other features could lead to even better results, so this is also an interesting direction of research.

IDE implementation. Implementing methods for single component prediction into an IDE, warning developers when they code or commit something potentially malicious, would help to prevent vulnerabilities before they even reach the final project code.

However, there are several threats that have a yet unknown impact. Researching them would help us to better understand how they affect the presented results, as well as providing us methods to increase the accuracy of this and other methods. Basically, the required tasks would be:

Chronological evaluation. This would require an evaluation model that respects the chronological order of vulnerabilities. From the results of such a model, we could verify that correlations also apply in this context and possibly how predictors change over the time.

Assessment of file groups affected by single vulnerabilities. It is still unclear how strong vulnerable component groups are connected to each other. If they can be separated, then the correlation between different vulnerabilities can be examined in a better way and the strength of the correlation between these and the correlation inside such a component group could be compared. Moreover, if the correlation in a component group affected by a single vulnerability is stronger, this could be used to point developers to related components during the process of fixing.

Bibliography

[1] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller. Predicting Vulnerable Software Components. *14th ACM Conference on Computer and Communications Security*, October 2007

[2] The Mozilla Foundation. *Mozilla project website. http://www.mozilla.org/*, January 2007.

[3] Larry Wall. The Perl programming language. *Perl directory. http://www.perl.org/*, 1987.

[4] The R Foundation. The R Project for Statistical Computing. *http://www.r-project.org/*, 2003.

[5] Vladimir Naumovich Vapnik. The Nature of Statistical Learning Theory. *Springer Verlag, Berlin*, 1995.

[6] E. Dimitriadou, K. Hornik, F. Leisch, D. Meyer, and A. Weingessel. Misc Functions of the Department of Statistics. TU Wien. *http://cran.r-project.org/src/contrib/Descriptions/e1071.html*, 2006.

[7] Brad Spengler. The grsecurity project. *http://grsecurity.net/.*

[8] The PaX Team. PaX patches. *http://pax.grsecurity.net/.*

[9] National Security Agency, Network Associates Laboratories, The MITRE Corporation and the Secure Computing Corporation. The SELinux Project *http://www.nsa.gov/selinux/*, 2001.

[10] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability 2006*, pages 25?33. ACM Press, October 2006.

[11] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In *Proceedings of the 15th Usenix Security Symposium*, pages 93-104, August 2006.

[12] M. Weber, V. Shah, and C. Ren. Cigital, Inc. A case study in detecting software security vulnerabilities usingconstraint optimization. In *Source Code Analysis and Manipulation, 2001. Proceedings.*, 2001

[13] J. Viega, J.T. Bloch, Y. Kohno, G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *16th Annual Computer Security Applications Conference (ACSAC'00)* acsac, page 257, 2000

[14] Jeffrey Voas and Gary McGraw. Software Fault Injection: Innoculating Programs Against Errors. John Wiley & Sons, 1997.

[15] Shuo Chen, Z. Kalbarczyk, J. Xu, R.K. Iyer. A data-driven finite state machine model for analyzing security vulnerabilities. *Proceedings of the Conference on the Dependable Systems and Networks*, 2003.

[16] B.P. Miller, L. Fredriksen, B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM 33, 12*, 1990.

[17] B.P. Miller. Fuzz Testing of Application Reliability. *http://pages.cs.wisc.edu/ bart/fuzz/fuzz.html*, 1990-2006.

[18] R. V. Hogg and A. T. Craig. *Introduction to Mathematical Statistics, 5th ed.* New York: Macmillan, pp. 338 and 400, 1995.

[19] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the 5th International Symposium on Empirical Software Engineering*, pages 18?27, New York, NY, USA, September 2006. Association for Computing Machinery, ACM Press.

[20] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference,and Prediction*, Chapter 13. Springer Series in Statistics. Springer Verlag, 2001.