Predicting Vulnerable Software Components

Stephan Neuhaus, Thomas Zimmermann, Andreas Zeller
Saarland University, Saarbrücken, Germany
{neuhaus, zimmerth, zeller}@cs.uni-sb.de

Abstract

We introduce Vulture, a new approach and tool to predict vulnerable components in large software systems. Vulture relates a software project's version archive to its vulnerability database to find those components that had vulnerabilities in the past. It then analyzes the import structure of software components and uses a support vector machine to learn and predict which imports are most important for a component to be vulnerable.

We evaluated Vulture on the C++ codebase of Mozilla and found that Vulture correctly identifies about two thirds of all vulnerable components. This allows developers and project managers to focus their testing and inspection efforts: "We should look at *nsXPInstallManager* more closely, because it is likely to contain yet unknown vulnerabilities."

1 Introduction

Suppose you are the development manager of a large application that is about to ship. Before release, you would certainly like to do your best such that the application does have a minimum of software vulnerabilities. Of course, "do your best" implies that your resources are limited. But still, you would probably like to spend your resources in the most effective way, finding (and fixing) the largest amount of vulnerabilities with the given resources. This means to focus your efforts those parts of the program which need it most—those components which have the highest risk of security vulnerabilities.

Allocating quality assurance resources is a difficult task. If you search for vulnerabilities in the wrong places, you waste time and money. If you do not search enough for vulnerabilities in components that are likely to contain some, a vulnerability may escape into production, with all consequences. What is needed, therefore, is a means to *predict where vulnerabilities are most likely to occur.*

Of course, such prediction takes place all the time. Developers and their managers remember failures as well as successes, and can build on their experience to direct their efforts. Furthermore, people can abstract from multiple incidents and come up with general rules that may help in identifying future problems. However, this requires that one actually *remembers* where past vulnerabilities were located; and the sheer number of incidents, not to speak of code or team size, makes it hard to remember all facts and abstract from them.

Since teams are aware of limitations of human memory, the typical approach to keep track of problems is to set up a *bug database*—a database which records all problems with the software. A bug database not only lists the current issues, but also the closed ones. Fixed problems can be linked to the fixes themselves—that is, *who* closed a bug by changing *which* component *when.*

Vulnerabilities form a specific subset of bugs. By mining the vulnerabilities from the bug database, we can learn which components are most prone to vulnerabilities. This is so because, as the vulnerabilities are fixed, the fixes apply to individual components—and therefore, one can actually compute how many vulnerabilities (or reported incidents) some component is associated with.

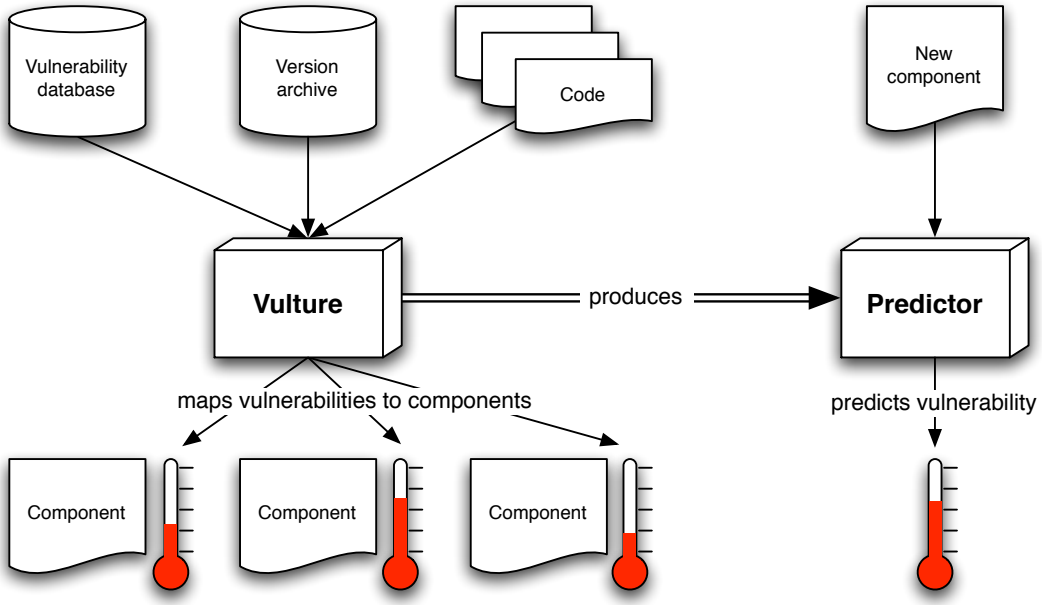

Figure 1: How Vulture works. Vulture mines a *vulnerability database* (e.g. a Bugzilla subset), a *version archive* (e.g. CVS), and a *code base*, and maps past vulnerabilities to components. The resulting *predictor* predicts the *future vulnerabilities of new components,* based on the structure of their imports.

For this paper, we have mined the security incidents from the Mozilla CVS and mapped them to individual components. As shown in Section 2, we found that of the 10,452 components, only 424 (or 4.05%) were involved in security fixes. Does this mean that managers only need to focus on these 4.05% of vulnerable components? Unfortunately, no—because the number of vulnerabilities in the past may not be indicative for the future:

- Software *evolves*, with substantial refactorings and additions being made all the time. This gradually invalidates earlier measurements about vulnerabilities.
- The vulnerabilities we measure from history can only be mapped to a component because *they have been fixed.* Thus, by definition, any measurement of vulnerable components applies to an already obsolete revision.

For these reasons, we need to come up with predictive methods that can be applied to new as well as evolved components—predictions that rely on *invariants* in domain and structure that allow predicting defects although the code itself has changed. What could be the nature of such invariants? In Section 3, we observe that the *domain*—as expressed by the other components that are interacted with—characterizes a component's vulnerability. In case of Mozilla, for instance, we found that of the 14 components importing *nsNodeUtils.h*, 13 components (93%) had to be patched because of security leaks. The situation is even worse for those 15 components that import *nsIContent.h*, *nsIInterfaceRequestorUtils.h* and *nsContentUtils.h* together, because they *all* had vulnerabilities. In other words: "Tell me what you import, and I'll tell you how vulnerable you are."

Would a new Mozilla component importing *nsNodeUtils.h* be prone to vulnerabilities as well? In Section 4, we develop statistical models that allow predicting whether a component—new or not—is vulnerable or not. In Section 5, we evaluate these models for Mozilla. Not only can we predict vulnerable components; we can even successfully predict their *vulnerability ranking*.

We have implemented this approach in a tool called *Vulture,* automating all the steps listed above. As sketched in Figure 1, Vulture parses given version and vulnerability databases, determines the most vulnerable components, and predicts the vulnerability of new components. Vulture is fully automatic: No interaction is required except for specifying the locations of the database and the CVS. This is the central contribution of this paper: A tool that *automatically learns from past vulnerabilities* to *predict the future vulnerability of new components.*

2 Components and Vulnerabilities

2.1 Components

For our purposes, a *component* is an entity in a software project that can have vulnerabilities. For Java, components would be *.java* files because they contain both the definition and the implementation of classes. In C++, and to a lesser extent in C, however, the implementation of a component is usually separated from its interface: a class is declared in a header file, and its implementation is contained in a source file. A vulnerability that is reported only for one file of a two-file component is nevertheless a vulnerability of the entire component. For this reason, we will combine equally-named pairs of header and source files into one component.

In C, it is often the case that libraries are built around abstractions that are different from classes. The usual case is that there is one header file that declares a number of structures and functions that operate on them, and several files that contain those functions' implementations. Without a working build environment, it is impossible to tell which source files implement the concepts of which header file. Since we want to apply Vulture to projects where we do not have a working build environment—for example because we want to analyze old versions that we cannot build anymore due to missing third-party software—, we simply treat files which have no equally-named counterpart as components containing just that file. We will subsequently refer to components without any filename extensions.

Of course, some components may naturally be self-contained. For example, a component may consist only of a header file that includes all the necessary implementation as inline functions there. Templates must be defined in header files. A component may also not have a header file. For example, the file containing a program's *main* function will usually not have an associated header file. These components then consist of only one file.

2.2 Mapping Vulnerabilities to Components

A *vulnerability* is a defect in one or more components that manifests itself as some violation of a security policy. Vulnerabilities are announced in security advisories that provide users workarounds or pointers to fixed versions and help them avoid security problems. In the case of Mozilla, advisories also refer to a bug report in the Bugzilla database. We use this information, to map vulnerabilities to components through the fixes that remove the defect.

First we retrieve all advisories from the Web to collect the defects, in case of Mozilla from the "Known Vulnerabilities in Mozilla Products" page.[1] We then search for references to the Bugzilla database that typically take the form of links to its web interface:

```
https://bugzilla.mozilla.org/show_bug.cgi?id=362213
```

The number at the end of this URL is the *bug identifier* of the defect that caused the vulnerability. We collect all bug identifiers and use them to identify the corresponding fixes in the version archive. In version archives

[1] http://www.mozilla.org/projects/security/known-vulnerabilities.html

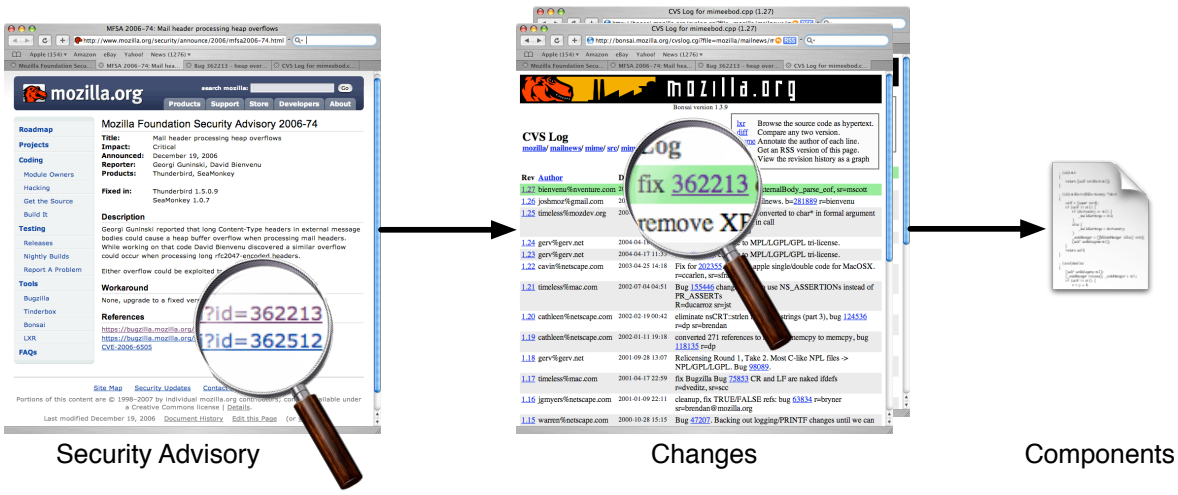

Figure 2: Mapping Mozilla vulnerabilities to changes. We extract bug identifiers from security advisories, search for the fix in the version archive, and from the corrected files, we infer the component(s) affected by the vulnerability.

every change is annotated with a message that describes the reason for that change. In order to identify the fixes for a particular defect, say 362213, we search these messages for bug identifiers such as "362213", "Bug #362213", and "fix 362213" (see also Figure 2). This approach is described in detail by Śliwerski et al. [28] and extends the approaches introduced by Fischer et al. [10] and by Čubranić et al. [7].

Once we have identified the fixes of vulnerabilities, we can easily map the names of the corrected files to components. Note that a security advisory can contain several references to defects, and a defect can be fixed in several files.

2.3 Vulnerable Components in Mozilla

Mozilla as of 4 January 2007 contains 1,799 directories and 13,111 C/C++ files which are combined into 10,452 components. There were 134 Mozilla Foundation Security Advisories, pointing to 302 bug reports. Of all 10,452 components, only 424 or 4.05% were vulnerable.

Since January 2005, security vulnerabilities in Mozilla are announced through Mozilla Foundation Security Advisories (MFSAs) [31]. These advisories describe the vulnerability and give assorted information, such as Bugzilla bug identification numbers. As of January 4, 2007, there were 134 MFSAs, pointing to 302 bug reports. Of these bug reports, 280 or 92.7% could be assigned to components using the techniques described above.[2]

It turns out that out of all 10,452 components, only 424 or 4.05% had vulnerability-related bug reports associated with them. We call these component *vulnerable*. In contrast to a vulnerable component, a *neutral* component has had no vulnerability-related bug reports associated with it so far.[3]

The distribution of the number of vulnerability-related bug reports and MFSAs can be seen in Figure 3. The top ten most vulnerable components in Mozilla are listed in Table 1. The three most vulnerable compo-

[2]Some bug reports in Bugzilla [30] are not accessible without an authenticated account. We suppose that these reports concern vulnerabilities that have high impact but that are not yet fixed, either in Mozilla itself or in other software that uses the Mozilla codebase. In many cases, we were still able to assign bug reports to files automatically because the CVS log message contained the bug report number. By looking at the diffs, it would therefore have been possible to derive what the vulnerability was. Denying access to these bug reports is thus largely ineffectual and might even serve to alert blackhats to potential high-value targets.

[3]Determining *invulnerable* components, i.e. those that will *never* have a vulnerability-related bug report associated with them, is beyond the scope of this paper.

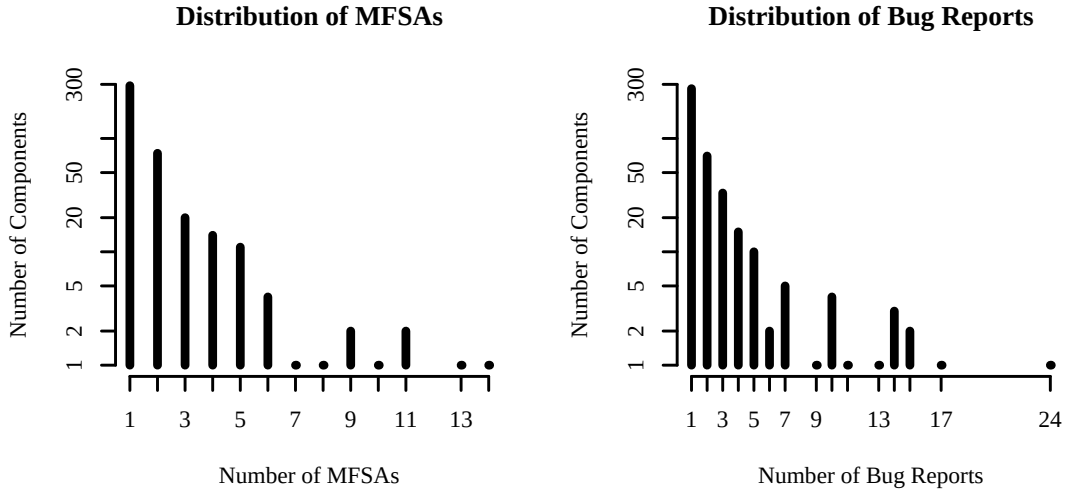

Figure 3: Distribution of Mozilla Foundation Security Advisories (MFSAs) and vulnerability-related bug reports in vulnerable components. The *y* axis is logarithmic.

Rank	Component	Directory	MFSAs	Bug reports
# 1	*nsGlobalWindow*	*dom/src/base*	14	14
# 2	*jsobj*	*js/src*	13	24
# 3	*jsfun*	*js/src*	11	15
# 3	*nsScriptSecurityManager*	*caps/src*	11	15
# 5	*jsscript*	*js/src*	10	14
# 6	*nsDOMClassInfo*	*dom/src/base*	9	10
# 7	*nsDocShell*	*docshell/base*	9	9
# 8	*jsinterp*	*js/src*	8	14
# 9	*nsGenericElement*	*content/base/src*	7	10
# 10	*nsCSSFrameConstructor*	*layout/base*	6	17

Table 1: The top ten most vulnerable components in Mozilla, sorted by associated MFSAs and bug reports.

nents all deal with *scripting* in its various forms:

1. *nsGlobalWindow,* with fixes for 14 MFSAs and 14 bug reports, has, among others, a method to set the status bar, which can be called from JavaScript and which will forward the call to the browser chrome.

2. *jsobj* (13 MFSAs; 24 bug reports) contains support for JavaScript objects.

3. *jsfun* (11 MFSAs; 15 bug reports) implements support for JavaScript functions.

In the past, JavaScript programs have shown an uncanny ability to break out of their jails, which manifests as a high number of security-related changes to these components. Not surprisingly, a security component, *nsScriptSecurityManager* with 11 MFSAs and 15 bug reports related to it is also ranked at position 3.

3 How Imports Matter

As discussed in Section 2.3, we found that several components related to *scripting* rank among the most vulnerable components. How does a concept like scripting manifest itself in the components' code?

Our central assumption in this work is that what a component does is characterized by its *imports*. A class that implements some form of content—anything that can be in a document's content model—will import *nsIContent.h*; a class that implements some part of the Document Object Model (DOM) will likely import *nsDOMError.h*. And components associated with scripting are characterized by the import of *nsIScriptGlobalObject.h*.

In a strictly layered software system, a component that is located at layer k would import only from components at layer $k + 1$; its imports would pinpoint the layer at which the component resides. In more typical object-oriented systems, components will not be organized in layers; still, its imports will include those components whose services it uses and those interfaces that it implements.

If an interface or component is specified in an insecure way, or specified in a manner that is difficult to use securely, then we would expect many components that use or implement that interface or component to be vulnerable. In other words, we assume that it is a component's *domain*, as given by the services it uses and implements, that determine whether a component is likely to be vulnerable or not.

How do imports correlate with vulnerabilities? For this, we first need a clear understanding of what constitutes an import and what it means for a set of imports to correlate with vulnerability.

3.1 Imports

C and C++, a component's *imports* are those files that it references through `#include` preprocessor directives.[4] They are handled by the preprocessor and come in three flavors:

`#include <name>` This variant is used to import standard system headers.

`#include "name"` This variant is used to import header files within the current project.

`#include NAME` This variant is a so-called computed include. Here, `NAME` is treated as a preprocessor symbol. When it is finally expanded, it must resolve to one of the two forms mentioned above.

Import extraction for C and C++ is difficult because the exact semantics of the first two variants are implementation-dependent, usually influenced by compile-time switches and macro values. That means that it is not possible to determine exactly what is imported without a working build environment. We adopted the following heuristics:

- We treat every occurrence of `#include` as an import, even though it may not be encountered in specific compile-time configurations—for example because of conditional compilation. The reason is that we want to obtain all possible import relations, not just the ones that are specific to a particular platform.
- For `<...>`-style includes, we assume that the name inside the angle brackets is not under the project's root directory. That means even if the compile-time switches are set so that, say, `#include <util.h>` is resolved to *src/util/util.h*, We will treat it as a reference to an external include file, different from *src/util/util.h*. it also means that if compile-time switches make `<util.h>` and `<util/util.h>` refer to the same file, we will treat them as two different imports.

[4] For Java projects, a component's imports would be referenced by `import` statements. Component extraction in Java is also much easier than for C and C++ since Java does not distinguish between the definition and implementation of a class and because everything has to be encapsulated in classes.

- For `"..."`-style includes, we assume that the name inside the double quotes is under the project root, even if compile-time switches might make them refer to different files, as above. If an include references `"util.h"`, but if there is no file called *util.h* under the project root, it is treated as a reference to an external include. If there are two or more files named *util.h*, the include will be treated as importing a "meta-include", consisting conceptionally of all like-named files.[5]

- Implementing the computed include would require a full preprocessor pass over the source file. This in turn would require us to have a fully compilable (or at least preprocessable) version of the project. Fortunately, this use of the include directive is very rare, so we chose to ignore it.

3.2 Mapping Vulnerabilities to Imports

In order to find out which import combinations are most correlated with vulnerabilities, we use frequent pattern mining [1, 17]. The result of frequent pattern mining is a list of import sequences that frequently occur in vulnerable components. To judge whether these imports are significant, we apply the following criteria:

Minimum Support. The pattern must appear in at least 3% of all vulnerable components. (In other words, it must have a minimum support count of 3% of 424, or 13).

Significance. We want to make sure that we only include patterns that are more meaningful than their sub-patterns. For this, we test whether the entire pattern is more specific for vulnerabilities than its sub-patterns. Let I be a set of includes that has passed the above test. Then for each proper subset $J \subset I$, we look at all files that import I and at all files that import $I - J$. We then classify those files into vulnerable and neutral files and then use the resulting contingency table to compute whether additionally importing J significantly increases the chance of vulnerability. We reject all patterns where we cannot reject this hypothesis at the 1% level. (In other words, it must be highly unlikely that including J in addition to $I - J$ is independent from vulnerability.)[6]

For patterns that survive these tests, the probability of it occurring in a vulnerable component is much higher than for its subsets. This is the case even though the conditional probability of having a vulnerability when including these particular includes may be small.

3.3 Imports in Mozilla

Again, we applied the above techniques to the Mozilla base. In Mozilla, Vulture found 79,494 import relations of the form "component x imports import y", and 9,481 distinct imports. Finding imports is very fast: a simple ANTLR parser [22] goes through the 13,111 C/C++ files in about two minutes.

Frequent pattern mining, followed by weeding out insignificant patterns yields 576 include patterns. The top ten patterns are shown in Table 2.

Going through all 576 include patterns additionally reveals that some includes occur often in patterns, but not alone. For example, *nsIDocument.h* appears in 45 patterns, but never appears alone. Components that often appear together with *nsIDocument.h* come from directories *layout/base* or *content/base/public*, just like *nsIDocument* itself.

[5] There are some additional disambiguation heuristics in place, and Vulture will do its best to find out the target of an include directive. For example, when a file includes `"util/util.h"`, and there is more than one file named *util.h*, but only one is in a subdirectory called *util*. However, describing all of them would be tedious.

[6] For this, we use χ^2 tests if the entries in the corresponding contingency table are all at least 5, and Fischer exact tests if at least one entry is 4 or less.

$P(V\|I)$	$V \wedge I$	$\neg V \wedge I$	Includes
1.00	13	0	*nsIContent.h · nsIInterfaceRequestorUtils · nsContentUtils.h*
1.00	14	0	*nsIScriptGlobalObject.h · nsDOMCID.h*
1.00	19	0	*nsIEventListenerManager.h · nsIPresShell.h*
1.00	13	0	*nsISupportsPrimitives.h · nsContentUtils.h*
1.00	19	0	*nsReadableUtils.h · nsIPrivateDOMEvent.h*
1.00	15	0	*nsIScriptGlobalObject.h · nsDOMError.h*
0.97	34	1	*nsCOMPtr · nsEventDispatcher.h*
0.97	29	1	*nsReadableUtils.h · nsGUIEvent.h*
0.96	22	1	*nsIScriptSecurityManager.h · nsIContent.h · nsContentUtils.h*
0.95	18	1	*nsWidgetsCID.h · nsContentUtils.h*

Table 2: Include patterns most associated with vulnerability. The column labeled "Includes" contains the include pattern; the column labeled $P(V|I)$ contains the conditional probability that a component is vulnerable (V) if it includes the pattern (I). The columns labeled $V \wedge I$ and $\neg V \wedge I$ give the absolute numbers of components that are vulnerable and include the set, and of components that are not vulnerable, but still include the set.

The table reveals that it implementing or using *nsIContent.h* together with *nsIInterfaceRequestorUtils* and *nsContentUtils.h* correlated with vulnerability in the past. Typical components that imports these are *nsJSEnvironment* or *nsHTMLContentSink*. The first is again concerned with JavaScript, which we already know to be risky. The second has had a problems with a crash involving DHTML that apparently caused memory corruption that could have led to arbitrary code execution (MFSA 2006-64).

It can also happen that a file that itself does not have a security leak is nevertheless changed in the course of a security fix. This happens, for example, when interfaces are changed and all classes that implement or use that interface also need to be changed.

Looking at Table 2, we see that of the 35 components importing *nsIScriptSecurityManager.h*, *nsIContent.h*, and *nsContentUtils.h*, 34 are vulnerable, while only one is not. This may mean one of two things: either the component is invulnerable or the vulnerability just has not been found yet. At the present time, we are unable to tell which is true. However, the component in question is *nsObjectLoadingContent*. It is a base class that implements a content loading interface and that can be used by content nodes that provide functionality for loading content such as images or applets. It certainly cannot be ruled out that the component has an unknown vulnerability.

4 Predicting Vulnerabilities from Imports

In order to predict vulnerabilities from imports, we need a data structure that captures all of the important information about components and imports (such as which component has which imports) and vulnerabilities (such as which component has how many vulnerabilities), but abstracts away information that we consider unimportant (such as the component's name). If there are m components and n imports, we write each component as a n-vector of imports: $\mathbf{x}_k = (x_{k1}, \ldots, x_{kn})$, where for $1 \leq k \leq m$ and $1 \leq j \leq n$,

$$x_{kj} = \begin{cases} 1 & \text{if component } i \text{ imports import } j, \\ 0 & \text{otherwise.} \end{cases}$$

We combine all components into $X = (\mathbf{x}_1, \ldots, \mathbf{x}_m)^t$, the project's *import matrix*. Entities that cannot import or that cannot be imported, such as documentation files, are ignored.

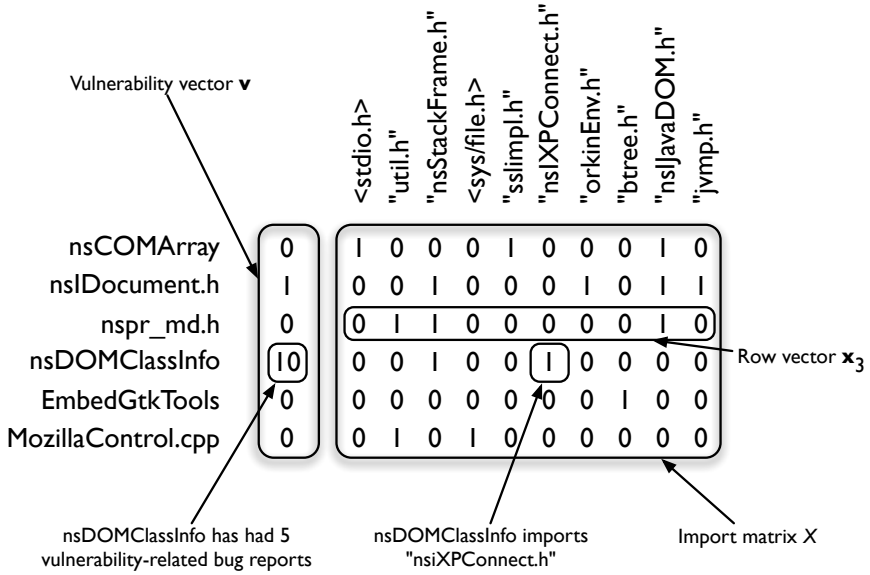

Figure 4: The import matrix X and the vulnerability vector $\mathbf{v}$. The rows of X contain the imports of a certain component as a binary vector: x_{ik} is 1 if component i imports import k. The vulnerability vector contains the number of vulnerability-related bug reports for that component.

In addition to the import matrix, we also have the *vulnerability vector* $\mathbf{y} = (y_1, \ldots, y_m)$, where y_j is the number of vulnerability-related bug reports associated with component j; see also Figure 4.

Now assume that we get a new component, $\mathbf{x}_{m+1}$. Our question, "How vulnerable is component $m+1$?" is now equivalent to asking for the rank of y_{m+1} among the values of $\mathbf{y}$, given $\mathbf{x}_{m+1}$; and our other question, "Is component $m+1$ vulnerable?" is now equivalent to asking whether $y_{m+1} > 0$.

As we have seen in the preceding sections, imports are correlated with vulnerabilities. How can we use this information to predict whether a new component will be vulnerable or not? And how can we predict whether a component will be more vulnerable than another so that we can better direct testing effort?

Both questions can be posed as machine-learning problems. In machine learning, a parameterized function f, called a *model*, is trained using training data X and $\mathbf{y}$, so that we predict $\hat{\mathbf{y}} = f(X)$. The parameters of f are usually chosen such that the error, that is, the difference between $\mathbf{y}$ and $\hat{\mathbf{y}}$, is minimized.

The question, "Is this component vulnerable?" is called *classification* (because it classifies components as vulnerable or not vulnerable), and "Is this component more or less vulnerable than another component?" can be answered with *regression*: by predicting the number of vulnerabilities and then ranking the components accordingly.

In our case, X would be the project's import matrix, and $\mathbf{y}$ would be the vulnerability vector $\mathbf{v}$. If we now train a model and feed it with a new component x' and if it classifies it as vulnerable, this means that this component has imports that were associated with vulnerabilities in other modules.

4.1 Validation Setup

To test how good imports work as predictors for vulnerabilities, we simply use our import matrix both to train and to assess the model. For this purpose, we randomly select a number of rows from X and the corresponding elements from $\mathbf{v}$—collectively called the *training set*—and use this data to train f. Then we use the left-over rows from X and elements from $\mathbf{y}$—the *validation set*—to predict whether the corresponding components are vulnerable and to compare the computed prediction with what we already know from the bug database. It is usually recommended that the training set be twice as large as the validation set, and we

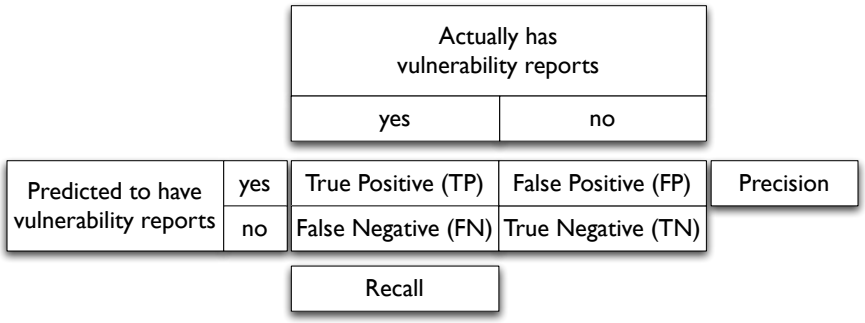

Figure 5: Precision and recall explained. Precision is $TP/(TP + FP)$; recall is $TP/(TP + FN)$.

are following that recommendation. We are not using a dedicated test set because we will not be selecting a single model, but will instead be looking at the statistical properties of many models and will thus not tend to underestimate the test error of any single model [13, Chapter 7].

One caveat is that the training and validation sets might not contain vulnerable and neutral components in the right proportions. This can happen when there are so few vulnerable components that pure random splitting would produce a great variance in the number of vulnerable components in different splits. This problem is solved by *stratified sampling*, which samples vulnerable and neutral components separately.

4.2 Evaluating Classification

For classification, we can now compare $\hat{\mathbf{y}}$ with $\mathbf{y}$ and count how many times our prediction was correct. This gives rise to the measures of *precision* and *recall*, as shown in Figure 5:

- The *precision* measures how many of the components predicted as vulnerable actually have shown to be vulnerable. A high precision means a low number of false positives; for our purposes, the predictor is *efficient.*
- The *recall* measures how many of the vulnerable components are actually predicted as such. A high recall means a low number of false negatives; for our purposes, the predictor is *effective.*

Achieving a maximum precision is easy—just predict *zero* components to be vulnerable. This implies maximum efficiency, but also a minimum recall. Likewise, achieving a maximum recall can be done by predicting *all* components to be vulnerable, which is effective, but not efficient, as the precision is minimal. The key is therefore in achieving a *balance* between precision and recall, or between efficiency and effectiveness.

4.3 Evaluating Ranking

When we use a regression model, we predict the *number* of vulnerabilities in a component. To assess the quality of the prediction, we want to know whether the *ranks* of the predicted and actual values correlate. This is because we will allocate our quality assurance resources to a component according to its rank: components that are ranked higher will get more resources.

To measure the quality of our rank prediction, we took the actual top 1% components and computed their predicted ranks; see Figure 6. We then used Spearman's rank correlation coefficient to measure the rank correlation. This is a real number between -1 and $+1$, where $+1$ means that the ranks agree perfectly, zero means that there is no correlation and -1 means that the ranks are in reverse order; see Figure 7.

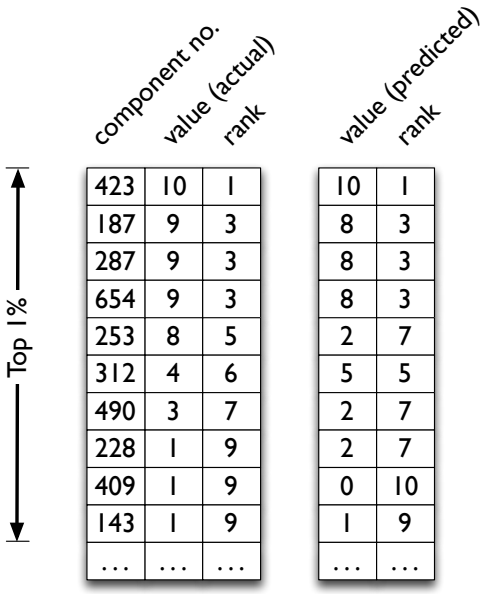

Figure 6: Rank correlation is computed for the top 1% of predicted values only. In this case, the rank correlation would be 0.94, which is very high. In the case of ties (equal values), we take the average rank.

4.4 Prediction using Support Vector Machines

For our model f, we chose support vector machines (SVMs) [34] over other models such as k-nearest-neighbors [13, Chapter 13] because they have a number of advantages:

- When used for classification, SVMs cope well with data that is not linearly separable.[7]
- SVMs come with plug-in kernels that can be used to achieve better effectiveness. Kernels add additional dimensions to the training data to achieve greater separability.
- SVMs have parameters that can be automatically tuned to achieve better effectiveness.

5 Case Study: Mozilla

To evaluate Vulture's predictive power, we applied it to the code base of Mozilla [32]. Mozilla is a large open-source project that has existed since 1998. It is easily the second most commonly used Internet suite (web browser, email reader, and so on) after Internet Explorer and Outlook.

5.1 Data Collection

We examined Mozilla as of January 4, 2007. Vulture imported the CVS and the MFSAs into a database, mapped vulnerabilities to components, and then created the input matrix and vulnerability vector as described in Sections 2.3 and 3.3.

Table 3 reports approximate running times for Vulture's different phases when applied to Mozilla. The only phases that would occur in a production environment would be the creation of the input matrix and vulnerability vector, and the computation of the SVM. Most of the time in importing the CVS into the

[7] Two sets of n-dimensional points are said to be *linearly separable* if there exists an $n-1$-dimensional hyperplane that separates the two sets.

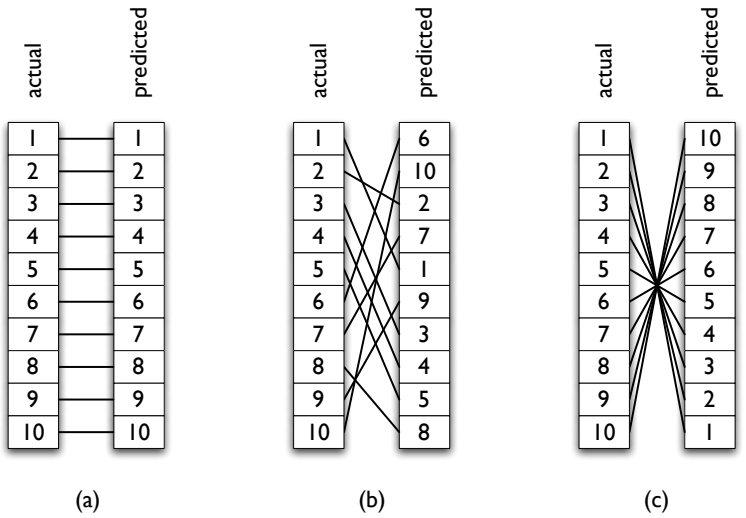

Figure 7: Rank correlation explained. Figure (a) has rank correlation $+1$, Figure (b) has rank correlation near 0, and Figure (c) has rank correlation -1.

Phase	Time
Importing CVS into database	24 h
Importing MFSAs into database, including download	5 m
Mapping vulnerabilities to components	1 m
Finding imports	2 m
Writing input matrix/vulnerability vector	5 m
Creation of SVM	15 m
Regression on validation set	3 m
Classification on validation set	2 m

Table 3: Approximate running times for Vulture's different phases.

database is spent reconstructing *transactions,* i.e. the logical coupling of check-ins of individual files. This step only occurs in CVS; it is not needed in more advanced configuration management systems such as Subversion, because they already support the concept of transactions. Also, importing the version archive and vulnerability information into a database would not occur in a production environment because either the source code control system or the vulnerability information would be already database-based or could be added to the database incrementally.

The 10,452 × 9,481 import matrix takes up 280 MB of disk space. From the import matrix and the vulnerability vectors, we created 40 random splits using stratified sampling. This ensures that vulnerable and neutral components are present in the training and validation sets in the same proportions. The training set had 6,968 entries and was twice as large as the validation set with 3,484 entries. Finally, we assessed these SVMs with the 40 validation sets.

5.2 Classification

For the statistical calculations, we used the R system [23] and the SVM implementation available for it [9]. It is very easy to make such calculations with R; the size of all R scripts used in Vulture is just about 200 lines. The calculations, however, used up to 11 GB of RAM.

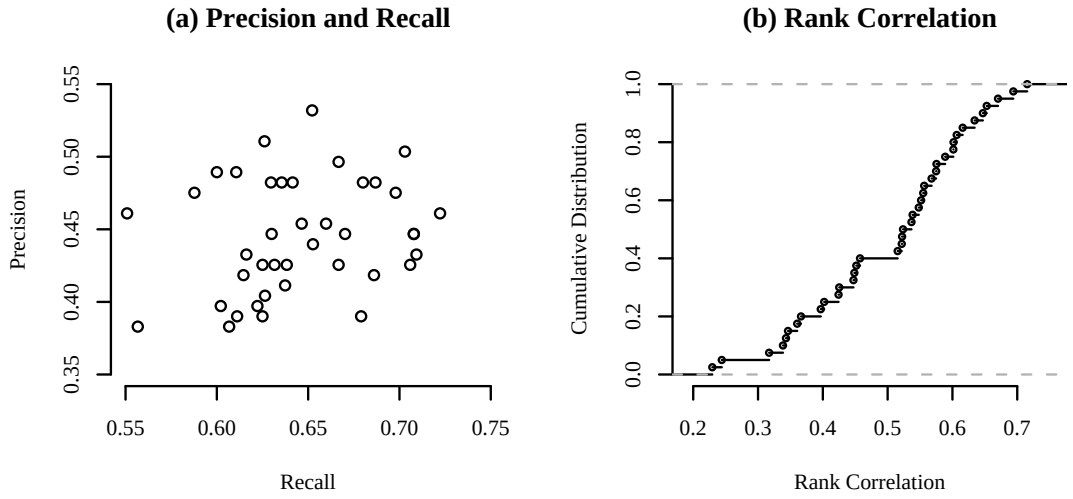

Figure 8: (a) Scatterplot of precision/recall values for the 40 experiments. (b) Plot of the empirical cumulative distribution function for the Spearman rank correlation coefficients of the 40 random splits. The x axis shows the Spearman rank correlation and the y axis shows the fraction of random splits where the correlation coefficient was less than that number.

Figure 8 (a) reports the precision and recall values for our 40 random splits. The recall has an average of 0.65 and standard deviation of 0.04, which means that two thirds of all vulnerable components are correctly classified:

Of all vulnerable components, Vulture flags 65% as vulnerable.

The precision has a mean of 0.45 and a standard deviation of 0.04, which means that a little less than half of the predictions turn out to affect components that have known vulnerabilities:

Of all components flagged as vulnerable, 45% actually are vulnerable.

5.3 Ranking

The rank correlation coefficients are shown in Figure 8 (b). The average rank correlation is +0.50 with a standard deviation of 0.12. The probability that the rank correlations are as high as they are, even when in fact there is no correlation, is less than 0.01 in 32 out of 40 cases, and less than 0.10 in all cases. This makes the results statistically significant.

There is a statistically significant fairly strong positive correlation between actual and predicted vulnerability ranks.

5.4 A Ranking Example

Let us illustrate the ranking correlation by an actual example. As a quality assurance manager, you want to focus extra efforts on the top ten new components that Vulture predicts as vulnerable. Table 4 shows such a prediction as produced in one of the random splits.[8] Within the validation set, these would be the components to spend extra effort on.

[8] We spent no effort in specifically selecting one of these splits.

Prediction		Validation set	
Rank	Component	Bug reports	Actual rank
# 1	*NsDOMClassInfo*	10	# 3
# 2	*SgridRowLayout*	1	# 95
# 3	*xpcprivate*	7	# 6
# 4	*Jsxml*	11	# 2
# 5	*nsGenericHTMLElement*	6	# 8
# 6	*Jsgc*	10	# 3
# 7	*NsJSEnvironment*	4	# 12
# 8	*Jsfun*	15	# 1
# 9	*NsHTMLLabelElement*	3	# 18
# 10	*NsHttpTransaction*	2	# 35

Table 4: The top ten most vulnerable components from a validation set, as produced by Vulture.

Your effort would be well spent, because *all* of the top ten components actually turn out to be vulnerable. (*SgridRowLayout* and *NsHttpTransaction* are outliers, but still vulnerable.) Furthermore, in your choice of ten, you would recall the top four most vulnerable components.

Focusing on highly ranked components further improves the hit rate.

5.5 Discussion

Our case study shows three things. First of all, allocating quality assurance efforts based on a Vulture prediction achieves a reasonable balance between effectiveness and efficiency. It is *effective* because two thirds of all vulnerable components are actually flagged. At the same time, Vulture is *efficient* because directing quality assurance efforts on flagged components yields a return of 45%—almost every second component is a hit. Focusing on the top ranked components will give even better results.

Furthermore, these numbers show that there is empirically an undeniable correlation between imports and vulnerabilities. This correlation can be profitably exploited by tools like Vulture to make predictions that are correct often enough so as to make a difference when allocating testing effort. Vulture has also identified imports that always (or very often) lead to vulnerabilities when used together and can so point out areas that should perhaps be redesigned in a more secure way.

Best of all, Vulture has done all this without the need to resort to intuition. This gives programmers and managers much-needed objective guidelines when it comes to allocating testing effort.

5.6 Threats to Validity

We have identified the following circumstances that could affect the validity of our study:

Study size. The correlations we are seeing with Mozilla could be artifacts that are specific to Mozilla. They might not be as strong in other projects, or the correlations might disappear altogether. From our own work analyzing Java projects, we think this is highly unlikely.

Bugs in the database or the code. The code that imports the CVS or the MFSAs into the database could be buggy; the import matrix and vulnerability vector could have been incorrectly calculated; or the R code that does the assessment could be wrong. All these risks were mitigated either by sampling small subsets and checking them manually for correctness, or by implementing the functionality a

second time starting from scratch and comparing the results. For example, the vulnerability matrix was manually checked for some entries, and the R code was rewritten from scratch.

Bugs in the R library. We rely on a third-party R library for the actual computation of the SVM and the predictions [9], but this library was written by experts in the field and has undergone cross-validation, also in work done in our group [27].

Wrong or noisy input data. The Mozilla source files could contain many "noisy" import relations in the sense that some files are imported but actually never used; or the MFSAs could accidentally or deliberately contain wrong information. Our models do not incorporate noise. From manually checking some of the data, we believe the influence of noise to be negligible, especially since results recur with great consistency, but it remains a (remote) possibility.

Yet unknown vulnerabilities. Right now, our predictions are evaluated against known vulnerabilities in the past. Finding future vulnerabilities in flagged components would improve precision and recall; finding them in unflagged components would decrease recall.

6 Related Work

Previous work in this area tried to reduce the number of vulnerabilities or their impact by one of the following methods:

Looking at components' histories. The Vulture tool was inspired by the pilot study by Schröter et al. [27], who first observed that imports correlate with failures. While Schröter et al. examined *general defects,* the present work focuses specifically on *vulnerabilities.* To our knowledge, this is the first work that specifically mines and leverages vulnerability databases to make predictions.

Evolution of defect numbers. Both Ozment at al. [21] as well as Li et al. [16] have studied how the numbers of defects and security issues evolve over time. Ozment et al. report a decrease in the rate at which new vulnerabilities are reported, while Li et al. report an increase. Neither of the two approaches allow mapping of vulnerabilities to components or prediction.

Estimating the number of vulnerabilities. Alhazmi et al. use the rate at which vulnerabilities are discovered to build models to predict the number of as yet undiscovered vulnerabilities [2]. They use their approach on entire systems, however, and not on source files. Also, in contrast to Vulture, their predictions depend on a model of *how* vulnerabilities are discovered.

Miller et al. build formulas that estimate the number of defects in software, even when testing reveals no flaws [19]. Their formulas incorporate random testing results, information about the input distribution, and prior assumptions about the probability of failure of the software. However, they do not take into account the software's history—their estimates would be unchanged no matter how large the history is.

Tofts et al. build simple dynamic models of security flaws by regarding security as a stochastic process [33], but they do not make specific predictions about vulnerable software components. Yin at al. [37] highlight the need for a framework for estimating the security risks in large software systems, but give neither an example implementation nor an evaluation.

Testing the binary. By this we mean subjecting the binary of the program in question to various forms of testing and analysis (and then reporting any security leaks to the vendor). This is often done with techniques like fuzz testing [18] and fault injection; see the book by Voas and McGraw [36].

Eric Rescorla argues that finding and patching security holes does not lead to an improvement in software quality [24]. But he is talking about finding security holes *by third-party outsiders in the finished product* and not about finding them *by in-house personnel during the development cycle*. Therefore, his conclusions do not contradict our belief that Vulture is a useful tool.

(Statically) examining the source. This usually happens with an eye towards specific vulnerabilities—for example, buffer overflows. Representative of the many static code scanners, we will briefly discuss ITS4, developed by Viega at al. [35]. Their requirement was to have a tool that is fast enough to be used as real-time feedback during the development process, and precise enough so that programmers would not ignore it. Since their approach is essentially pattern-based, it will have to be extended as new patterns emerge. Since ITS4 checks local properties, it will also be very difficult for it to find security-related defects that arise from the interaction between far-away components, that is, components that are connected through a long chain of def-use relations. Also, ITS4, as it exists now, will be unable to adapt to programs that for some reason contain a number of pattern-violating but safe practices, because it completely ignores a component's history.

Another approach is to use model checking [3, 4]. In this approach, specific classes of vulnerabilities are formalized and the program model-checked for violations of these formalized properties. The advantage over other formal methods is that if a failure is detected, the model checker comes up with a concrete counter-example that can be used as a regression test case. This too is a useful tool, but like ITS4, it will have to be extended as new formalizations emerge. Some vulnerability types might not even be formalizable.

Vulture also contains a static scanner—it detects imports by parsing the source code in a very simple manner. However, Vulture's aim is not to declare that certain lines in a program might contain a buffer overflow, but rather to direct testing effort where it is most needed by giving a *probabilistic assessment* of the code's vulnerability.

For other approaches, see for example linear programming [12], data-flow analysis [14], locating functions near a program's input [8][9], checking against an axiomatization of correct pointer usage [11], exploiting semantic comments [15], symbolic bounds checking [25] symbolic pointer checking [26], or computing and checking path conditions [29].

Hardening source or runtime environment. By this we mean all measures that are taken to mitigate a program's ability to do damage. Hardening a program or the runtime environment is useful when software is already deployed. StackGuard is a method that is representative of the many tools that exist to lower a vulnerability's impact [6]. Others include mandatory access controls as found in AppArmor [5] or SELinux [20]. However, Vulture works on the other side of the deployment divide and tries to direct programmers and managers to pieces of code requiring their attention, in the hope that StackGuard and similar systems will not be needed.

7 Conclusions and Future Work

We have introduced Vulture, a new tool that predicts vulnerable components by looking at their imports. It is fast and reasonably accurate: it analyzes a project as complex as Mozilla in about half an hour, and correctly identifies two thirds of the vulnerable components. Half of its predictions are correct.

[9] The hypothesis of DeCast et al. that vulnerabilities occur more in functions that are close to a program's input is not supported by the present study. Many vulnerable components, such as *nsGlobalWindow*, lie in the heart of the application.

The contributions of the present paper can be summarized as follows:

1. A technique for locating past vulnerabilities by mining and combining vulnerability databases with version archives.
2. A tool that learns from the locations of past vulnerabilities to predict future ones with reasonable accuracy.
3. An approach for identifying vulnerabilities that automatically adapts to specific projects and products.
4. A predictor for vulnerabilities that needs only import structures, and thus can be applied before the component is fully implemented.

Despite these contributions, we feel that our work has just scratched the surface of what is possible, and of what is needed. Our future work will concentrate on the following topics:

Characterizing domains. We have seen that empirically, imports are good predictors for vulnerabilities. We believe that this is so because imports characterize a component's domain, that is, the type of service that it uses or implements, and it is really the domain that determines a component's vulnerability. We plan to test this hypothesis by studies across multiple systems in similar domains.

Fine-grained approaches. Rather than just examining imports at the component level, one may go for more fine-grained approaches, such as caller-callee relationships. Such fine-grained relationships may also allow vulnerability predictions for individual classes or even individual methods or functions.

More significant features. Besides imports, there may be further characteristics that make a component vulnerable. We plan to examine further features, such as complexity metrics, or the usage of specific data types; and to check whether such features would be applicable to predict vulnerabilities.

Evolved components. This work primarily applies to predicting vulnerabilities of new components. However, components that already are used in production code come with their own vulnerability history. We expect this history to rank among the best predictors for future vulnerabilities.

Usability. Right now, Vulture is essentially a batch program producing a textual output that can be processed by spreadsheet programs or statistical packages. We plan to integrate Vulture into current development environments, allowing programmers to query for vulnerable components. Such environments could also visualize vulnerabilities by placing indicators right next to the individual entities (Figure 9).

Vulture is part of the "Mining software archives" project at Saarland University. For more information, see

`http://www.st.cs.uni-sb.de/softevo/`

Acknowledgments

We thank the Mozilla team for making their databases available. David Schuler and Andrzej Wasylkowski provided constructive feedback on earlier revisions of this paper. Thomas Zimmermann is funded by the DFG-Graduiertenkolleg "Leistungsgarantien für Rechnersysteme".

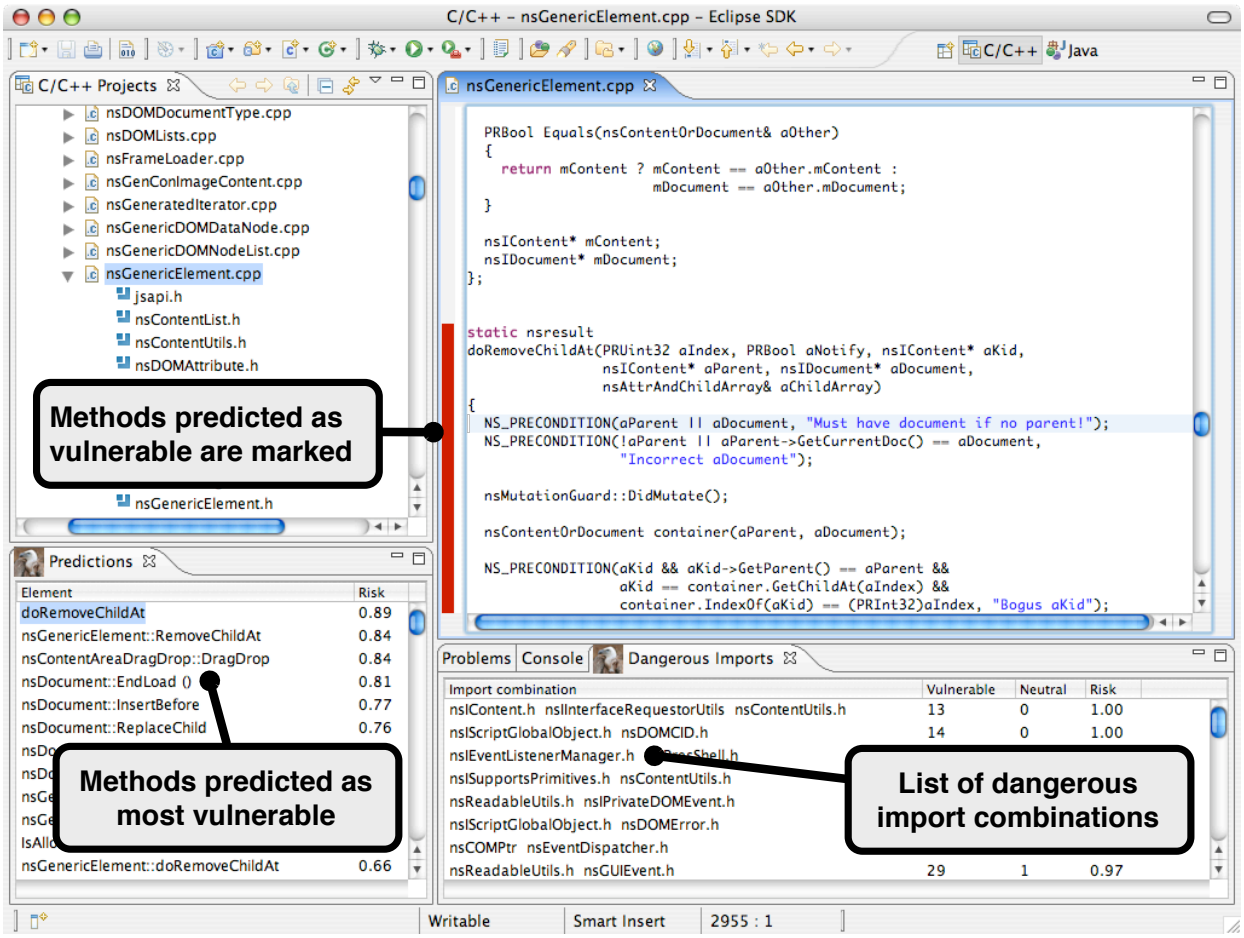

Figure 9: Sketch of a Vulture integration into Eclipse. Vulture annotates methods predicted as vulnerable with red bars. The view *"Predictions"* lists the methods predicted as most vulnerable. With the view *"Dangerous Imports"*, a developer can explore import combinations that lead to past vulnerabilities.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, September 1994.

[2] Omar Alhazmi, Yashwant Malaiya, and Indrajit Ray. *Security Vulnerabilities in Software Systems: A Quantitative Perspective*, volume 3645/2005 of *Lecture Notes in Computer Science*, pages 281–294. Springer Verlag, Berlin, Heidelberg, August 2005.

[3] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 171–185, February 2004.

[4] Hao Chen and David Wagner. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, November 2002.

[5] Crispin Cowan. Apparmor linux application security. `http://www.novell.com/linux/security/apparmor/`, January 2007.

[6] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

[7] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, June 2005.

[8] Dan DaCosta, Christopher Dahn, Spiros Mancoridis, and Vassilis Prevelakis. Characterizing the security vulnerability likelihood of software functions. In *IEEE Proceedings of the 2003 International Conference on Software Maintenance (ICSM'03)*, September 2003.

[9] Evgenia Dimitriadou, Kurt Hornik, Friedrich Leisch, David Meyer, and Andreas Weingessel. *e1071: Misc Functions of the Department of Statistics (e1071), TU Wien*, 2006. R package version 1.5-13.

[10] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM'03)*, Amsterdam, Netherlands, September 2003. IEEE.

[11] Pascal Fradet, Ronan Caugne, and Daniel Le Métayer. Static detection of pointer errors: An axiomatisation and a checking algorithm. In *European Symposium on Programming*, pages 125–140, 1996.

[12] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *10th ACM Conference on Computer and Communications Security (CCS)*, October 2003.

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

[14] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2006.

[15] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*, pages 177–190, August 2001.

[16] 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.

[17] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In *Knowledge Discovery in Databases: Papers from the 1994 AAAI Workshop*, pages 181–192, 1994.

[18] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

[19] K.W. Miller, L.J. Morell, R.E. Noonan, S.K. Park, D.M. Nicol, B.W. Murrill, and M. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE Transactions on Software Engineering*, 18(1):33–43, January 1992.

[20] National Security Agency. Security-enhanced linux. http://www.nsa.gov/selinux/, January 2007.

[21] 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.

[22] Terence Parr. ANTLR reference manual. http://www.antlr.org/doc/index.html, January 2007.

[23] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0.

[24] Eric Rescorla. Is finding security holes a good idea? *IEEE Security and Privacy*, 3(1):14–19, 2005.

[25] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 182–195. ACM Press, 2000.

[26] Berhard Scholz, Johann Blieberger, and Thomas Fahringer. Symbolic pointer analysis for detecting memory leaks. In *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 104–113. ACM Press, 1999.

[27] 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.

[28] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the Second International Workshop on Mining Software Repositories*, pages 24–28, May 2005.

[29] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. In *Proceedings of the 24th International Conference on Software Engineering*, New York, NY, USA, May 2002. ACM Press.

[30] The Mozilla Foundation. Bugzilla. http://www.bugzilla.org, January 2007.

[31] The Mozilla Foundation. Mozilla foundation security advisories. http://www.mozilla.org/projects/security/known-vulnerabilities.html, January 2007.

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

[33] Chris Tofts and Brian Monahan. Towards an analytic model of security flaws. Technical Report 2004-224, HP Trusted Systems Laboratory, Bristol, UK, December 2004.

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

[35] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. Token-based scanning of source code for security problems. *ACM Transaction on Information and System Security*, 5(3):238–261, 2002.

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

[37] Jian Yin, Chunqiang Tang, Xiaolan Zhang, and Michael McIntosh. On estimating the security risks of composite software services. In *Proceedings of the PASSWORD Workshop*, June 2006.