Classifying the Data Semantics of Patches

University of Calgary Technical Report 2013-1047-14

Robin Gonzalez Michael E. Locasto

September 3, 2013

Abstract

Patching software remains a key defensive technique for mitigating flaws and vulnerabilities. Patches, however, entail complications that are hard to predict. Patches can be incomplete or incorrect, thereby not fully addressing the targeted flaw or introducing new bugs and unintended behavior. System administrators and owners are often at a loss to assess the risk that applying a patch might carry. Without a lengthy evaluation, they cannot predict how the patch will behave in or affect their environment. Such obstacles often prevent the use of hot patching or dynamic software updating. One major obstacle to hot patching arises from the desynchronization of existing data with the patch's new code semantics.

This paper adopts a machine learning approach to assist this kind of prediction: whether the patch contains elements that are likely to cause problems if the patch is applied to the running system. We drive this automated assessment (based on a Support Vector Machine) via an analysis of the control and data modification operations in the patch. Our SVM classifies a set of 25 unlabeled patches with 92% accuracy. As a baseline, it also classifies its testing set of 50 patches (blindly, without labels) with 84% accuracy.

1 Introduction

Organizations and end users need patch triage. More precisely, they need help performing such procedures due to the volume and complexity of patches. We posit that some form of automated classification would be useful to prioritize the testing and rollout of security patches. Such an automated facility would be *especially* useful for hot patching (i.e., dynamic software updates). One challenge for hot patching is the question of data structure updates. If a code patch introduces new semantics to a variable or data structure, the "live", in–memory data might become incompatible with the new semantics of the modified code. Software vendors typically require an application restart to bypass this problem, trading off correctness against availability.

Accordingly, we constrain our focus to the question of how best to augment users' ability to deploy security patches quickly (i.e., without a restart); this paper does not address the broader questions involved in non-security patches or offline patching. Therefore, we construct a machine learning technique for triaging the "feasiblitliy" of applying a hot patch based on features relevent to data modifications. Our proposed solution takes the form it does for two reasons: (1) we employ machine learning to achieve a certain amount of automation and (2) our SVM classifies patches based on data modifications because we focus on data semantics of hot patching as the key obstacle to the widespread use of hot patching.

1.1 Stuck With Patching

Patching remains an important part of "practical" system security, but users often avoid patching for a few reasons.

First, patching implies restarting the application: addressing vulnerabilities in this way is tedious, slow, and inconvenient. Patches disrupt a user's current working context because they often require an application restart or OS reboot. Despite the introduction of concepts like microrebooting [6], rebooting for real software still represents substantial downtime rather than a smooth transition. Patches rarely occur at convenient times, and a standard schedule (i.e., Patch Tuesday) does little to rectify this unpleasant reality. Planned downtime, redundancy, and rolling updates may work for some large, well-run organizations (e.g., Google or other cloud SaaS vendors), but holds little comfort for busy end-users, small businesses, or underfunded organizations.

Since patching often holds as much risk (and little immediately evident downside) as remaining unpatched, users tend to delay their attention to this protection mechanism, as a study of Firefox update patterns by Frei *et al.* [14] suggests:

> "We conjecture that users in large part do not actually patch their Web browsers based on recommendations, perceived threats, or any security warnings...the Firefox update dynamics measurements revealed that despite the single click integrated auto-update functionality, rather surprisingly, one out of five Firefox

users surfs the Web with an outdated browser version."

Second, the implications of applying a patch are difficult to assess *a priori*, and vendors leave end-users to their own devices by providing little or no support for automated testing of the patch. It is difficult to predict what the future behavior of a patched application will be like [19,24,34]. Patches and updates have the potential to change system behavior in unanticipated ways (for example: a Norton patch [20], a McAfee [22] update). Furthermore, the "patches for patches for patches" problem (the initially incorrect fetchmail [1] fix is but one example of this problem that plagues both proprietary and open source vendors) shows that patching is far from an exact science. Many patches cause conflicts once applied, resulting in a cascading series of repatching [9,13,25].

System owners must therefore test patches before deployment. **In the case of end-users, determining the potential ill effects of a patch is therefore an exercise left to those least well–equipped to reason about the patch and the environment it enters.** For most users, it does not matter whether the patch is a source code patch to an open source program or a heavily obfuscated binary patch to a proprietary application. Users still face either a substantial reverse-engineering of the patch or an extensive software regression testing effort (or both).

1.2 Goal: When is it Safe to Hot Patch?

Although work in the field of software self–healing and dynamic software updates (DSU) has attempted to apply patches to a running software program as a reactive form of protection (with the hope of eliminating downtime due to restarts or reboots), relatively little progress has been made in mainstreaming these approaches [5,8,10,23,27,33,36].

When patching an application, restarting it is usually a necessity because of outdated states of data or control. Thus, in a DSU system, after patching an application, the old state of the program needs to be synchronized with the new semantics.

As an example, if we apply this simple patch to an application:

```
+ if (var > 10)
+     var2 = 0;
```

we are not only patching control flow, but we are also changing the semantics of some data structure in the program. The patch can also be read as *from now on, whenever var is bigger than 10, make var2 equal to 0.*

To the best of our knowledge, few DSU systems deal effectively with this semantics mismatch [12]. A patch may contain a different set of assumptions about the type, state, and structure of existing data structures and variables. What DSU–style systems seem to lack is a way to predict whether or not the patch contains operations that will cause a mismatch. This paper proposes one such prediction method.

1.3 Contributions

Our work provides evidence supporting the notion that patches can be classified according to feasibility (in terms of a feature vector based on properties of their data modifications and control flow). We can use that classification as an indication of the difficulty of applying them in a hot patch context. To the best of our knowledge there exists no structured study of how data is modified by a large collection of security patches.

1. We present a study of 75 different security patches; we translate them to a structured graph syntax and then to a feature vector derived from that graph language. The feature vector contains important attributes of each patch (Section 3 explains the features and their derivations).
2. We define a data patch language as a set of primitive algebraic operations. We manually analyze each patch in our dataset and determine how many patches with data structures are feasible to be updated using these heuristics. The purpose of this manual analysis is to establish ground truth for this dataset; the manual analysis is not a requirement for the SVM technique.
3. We process five randomly–selected subsets of our dataset through our SVM and obtain a breakdown of how many of these patches are feasible according to their features. We then compare these results with our manual "ground truth" classification of patches. We believe our methodology is novel and helps provide some insight in a space that has generally been lacking in quantitative analysis.
4. Finally, we look at two other randomly–selected subsets of our dataset and we offer an analysis of the relationship between the features from our feature vector and a patch being "feasible" or "infeasible" according to our SVM. This discussion helps provide insight into the often opaque models that SVMs produce. We examine the features that feasible patches have in common.

1.4 Definitions

Hot Patching: Dynamically updating an application by modifying both code and state without restarting the program.
Data Patch: The translation of a set of statements that modify data structures into a set of basic algebraic operations that consist of: {read, write, search, and compare} – simple database operations. A data patch is a group of statements that uses primitive relational algebraic operations (think of these operations as the ones that we can perform over data in a database) to correctly patch the data with the new semantics that the code patch implies.
Feasibility: We consider a patch to be feasible when, after hot patching it, the application does not have any conflicts between code semantics and data structures. See Section 2.2 and Section 3.1 for the semantics of "feasibility" as used in this paper.

2 Approach

Our workflow consists of seven main stages. We provide a very brief enumeration of this workflow here and follow it with a high–level explanation of the major steps. We expand on the details in later sections.

1. We gathered 100 patches from open source projects like Apache, Asterisk, and Samba. All of these patches are in the C programming language and were released between 2001 and 2012. We discarded 25 patches because they represented non–security patches or contained a number of combined updates whose joint size would have frustrated manual analysis (to establish ground truth). For our first experiment, we divided the remaining 75 patches into two groups:
    - **Training group:** We selected 50 patches that represented our training data. This data works as an input, in step 5, for the SVM and includes a label categorizing the patch as infeasible or feasible according to our definition of feasibility.
    - **Testing group:** We also selected 25 patches that represented the data for testing purposes. We give the data to the already trained SVM and get a result of feasible or infeasible according to the SVM.

   Our second experiment divided the set of 75 patches into five groups of 30 patches (training) and 45 patches (testing).

2. We manually translated each patch of the training and testing datasets into their graph representation, according to the heuristics defined in Section 3.
3. We translate each graph into an initial feature vector that represents all of the attributes, or features, from each graph.
4. We apply PCA over the initial feature vectors to establish correlation. PCA supplies the most independent combinations of features; we incorporate the most significant into our final feature vector.
5. We manually trace each of the 75 patches to classify them according to our definition of feasibility. This consisted of a careful and detailed analysis of each statement in the patch and with reference to the full source code.
6. By having a final feature vector of all the patches in the training and testing datasets, we give the training feature vectors as input to our SVM in the form of a matrix. Each vector includes a label classifying it as "feasible" or "infeasible" according to step 5.
7. We then apply our testing and training datasets, without any labels, as a feature vector matrix. Our SVM gives a prediction on feasibility according to what it learned.

2.1 Overview

To classify patches, we first need a way to represent patches as input for a machine and choose the best machine learning technique for classifying. Other authors [29] have used machine learning techniques, such as Support Vector Machines (SVMs), over source code by having an intermediate representation of the source code as graphs. We take a similar approach by translating the control flow of a patch into a graph, where the leaves of each subtree represent data modifications. We then extract the features of each graph into a feature vector for our SVM. We decided to use SVMs instead of many other machine learning techniques, because it is often used to classify source code by using machine learning techniques.

2.2 Feasible and Infeasible

We use the term "feasibility" to mean **being able to correctly update every data structure in an application according to the new semantics implied by a patch**. In other words, if we are able to translate a set of operations, that are modifying data structures, into a data patch then we consider the patch to be feasible to implement as a hot patch. If a patch is found to be feasible then it means that the application does not have any conflicts within its data structures after hot patching the patch. In other words, we are able to update the data structures of the application according to what we parsed from the patch.

Our definition of "feasibility" is therefore a binary decision that results from a deep manual analysis of the patch in context with the source code it is patching and knowledge of the live data structures. Obviously, such manual analysis is time–consuming and demands expertise in binary analysis and memory forensics on a per–application basis. We undertake this analysis for all the patches in our data set for the purpose of establishing ground truth. We do not, however, incorporate manual analysis into the SVM decisions involved in our experiments: our intent is to assess how well a machine learning approach might approximate an expert human decision.

We make a distinction between "expressibility" and "feasibility": the former term relates to whether an operation implied by a patch can be translated to our data patch mini–language and the latter refers to whether or not the data patch as a whole will successfully patch a running application. For a patch statement to be *expressible* in our data patch language, we need to be able to translate it into simple data manipulation operations (we define this set as {search, read, compare and write }). However, we are not able to express every statement using these operators. We need more computational power than what these statements offer in order to express every possible statement in a patch. However, we can model many functions and operators that do not involve arbitrary computation (in keeping with the principles of langsec [1]).

Most of the patches we label "infeasible" received that label due to user input, significant changes of the control flow (e.g. calling a function that calls another function), loops that recursively call functions and modify a variable according to the return value of the function or undecidable computation (e.g. modifying the if case condition but not the data within the if case scope).

2.3 Using Support Vector Machines

SVMs are a supervised machine learning technique (i.e. we used labeled data to train the machine, specifying whether or not the patch was feasible). The training dataset defines a hyperplane that separates feasible from infeasible patches.

[1]`http://langsec.org`

The focus of our study is whether there exists a relationship between the set of features for each graph and our definition of feasibility for a patch. More precisely, after applying our SVM and defining a hyperplane – which is the relationship of the feature vectors of our training data – and getting positive and negative data points by labeling the data set with **feasible** or **infeasible**, we want to study a possible relationship between the features and feasibility, and see if the machine can classify arbitrary patches by taking their feature vectors as input.

Why should we use Machine Learning algorithms? To investigate the applicability of transforming security patches into a form suitable for hot patching, a first approach might manually analyze each patch and attempt to express it as a "data patch." This is a tedious and error-prone process which consists of tracing the control flow of each patch. An automated approach could take advantage of the speed and scale of machine learning techniques to classify these patches by first having a small subset of patches already classified with a corresponding label of "feasible" or "infeasible."

2.4 Features of a Patch

We chose what we considered to be the most important features for the patch graphs, which included their cyclomatic complexity, computational cost, and number of primitive operations, among others. We also included important features of a data patch, such as number of write, search, read and compare operations, function calls returning variables, number of operations inside a loop, etc.

Each graph represents the control flow of a patch according to its data modifications. The leaves in each graph are the data modifications, and each subtree represents the way these data modifications are stated by the patch. We can think of this feature vector as the complexity of a patch, which is related to the control flow of its data modifications.

2.5 Labeling

This labeling activity is a prime example of the difficult and time–consuming nature of *manual* patch analysis and classification (precisely the obstacle our automated ML approach seeks to overcome). This labeling activity limited our research effort to some extent; translating each patch into its feature vector form could be automated, but the process of labeling a patch for the supervised learning process and verifying the results of our testing dataset could not be automated a priori. This process requires us to trace the source code of each patch in the training data set and see if we are able to express every function and library call. This is a non-deterministic process, and there is no algorithm to determine this.

3 Translation of Patches

This section explains the details of our translation procedure from a set of C language statements to a graph description language. Previous work on applying machine learning techniques on source code [35] takes a similar translation approach, and other graph techniques are also used for virus detection [18]. Since feature vectors are suitable as input to SVMs, we need to translate the statements in patches to some intermediate representation. We chose a graph–based representation to serve two purposes (besides having a rich natural set of features to extract). First, it was natural to model the relationships of patch statements as elements of a syntax tree. Second, we could mark up the tree with annotations describing the primitive relational algebraic operations (i.e. search, compare, read and write) to compute the total cost of each patch when translated to our language. We then use total cost of computation as a feature of a vector.

To create the graph, we model a patch as a collection of data modification statements interacting with the new heuristics of the patched application. Each statement is represented as a node in our graph description language. The control flow, represented as subtrees, explains how these statements interact with each other. The leaves of the graph represent data modifications and each subtree represents a different data modification statement.

The features of a graph – at least before applying our SVM – do not define feasibility but rather they define the control flow of data modifications in a patch. In order to define the feasibility of a patch, we must manually analyze the patch and classify it as feasible or infeasible depending on whether we are able, or not, to express data modifications using relational algebraic operations. These **primitive relational algebraic operations** are:

1. **Search:** A primitive operation that finds a variable/data structure instance/member of a data structure in a database. We are able to use operations such as search(name of var) or search(type of var). Because we also assume we have metadata such as: address, type, size and value, we are also able to express some operators such as sizeof()

or addressof() – the & operator on C – and functions such as strlen() and free() [2].

2. **Write:** It is a primitive relational algebraic operation that stores a value in an already searched variable or data structure. It is usually called after searching for a particular variable. Our heuristics also allow to pass parameters such as the size, type and address as metadata of the variable.
3. **Read:** Similarly to the operation *write*, this operation is commonly used after searching for a variable. It allows the application to read the value of an variable and apply some heuristics on it. Most of the time, patches use read operations to compare variables to values or write values on variables to new variables.
4. **Compare:** The compare operation is usually used in branches and loops. It denotes a comparison between a variable and a value or another variable.

3.1 Feasible and Infeasible Patches

Not all statements are translatable to these operations. We need more computational power than what these statements offer in order to express every possible statement in a patch. However, we can model many functions and operators that do not involve arbitrary computation. Some examples of what we consider to be infeasible statements in a patch are:

1. **Arbitrary function calling:** We are not able to express function calls if the data returned by the function depends on an input value.
2. **Undecidability:** Some statements are undecidable because it is impossible to construct an algorithm with a correct response to the outcome of the statement.
3. **Function and library calls:** We are able to model most of the function and library calls, however this requires us to manually trace the source code to model it.
4. **Macros:** We are not able to deduct macros defined by the developer. However, by looking at the source code, we are able to know the value of the macro and substitute it in our data patch.
5. **Loops:** We are able to express loops using iterations, but this is computationally expensive. Languages with a relational algebraic ap-

[2]This function is the equivalent of eliminating the value on a particular address, more precisely, by giving the address make the value of the variable to be NULL

proach, as SQL, however, also offer procedural languages to define loops and other operations, expanding our set of operations.

6. **Variables that are not currently in scope:** These variables are not able to be modified because they do not exist in that particular moment. However, these variables do not affect the heuristics of the program, therefore we do not need to patch them.

3.2 Data Modification Statements

We can also see a data modification machine as the combination of statements that ultimately modify some data structure. It is intuitive to think of data modifications as operations in a database, thus we can express them as a collection of simple relational algebraic operations. We show how to represent different data modifications using primitive operations such as search, read, write and compare. We also define a graph description that represents each machine.

Data modifications are the main operation we look for when translating patches into graphs. It can be seen as the leaves of the graphs and it is the most basic operation our model can do. We represent these machines as a $\bullet$ in graphs and as the operations *search and write* in our language. The computational cost of this operation is of 2.

Variable declaration – some patches add new variables to the application by declaring, and sometimes initializing, them as part of the new semantics of the patch. This operation can also be represented in our model by the operation *write*. Because there is no need for searching for a variable that does not exist, the computational cost of this operation is only 1.

Branches – We are also able to express relational operations as *comparisons* when reading or writing data. Therefore, we are also able to model **if** and *else* cases. For the purposes of this work we care for cases that involve data modifications.

We can express **loops** using iterations of primitive operations, however this could be computationally expensive. As it was stated before, some relational algebraic languages offer a procedural language that allows us to model loops, so we do not think of loops as a limitation. However, most of the patches that had loops were considered to be infeasible because they change the control flow of a patch by introducing recursion of functions and modifying variables by using their returning values.

We can express a limited number of other operators in C such as selected functions, including:

1. String Length: By getting the size of a character array. We are not able to deduce character pointers.
2. Address of, typeof, sizeof: Metadata of each variable.
3. Free: By setting the value of the variable on a particular address to NULL.

3.3 Constructing Graphs From Patches

In order to build a graph from a patch, we first represent the control flow of the patch according to its data modifications. Graphs are represented as trees, and the leaves of these trees are specific data modifications. In Table 1, we can see some of the graphical representation of the types of data modifications that are taken into consideration. We model patches as a collection of machines that interact with each other to fulfill the purpose of patching data.

Algorithm 1 Patch 1 example

```
procedure DataPatch1
    int $var = -1$
    if $var < 10$ then
        if $var2 < 10$ then
            $var2 = 8$
        else $finish$
        end if
    else $finish$
    end if
    if $var < 5$ then
        if $var2 < 2$ then
            free($var2$)
            $var2 = 10$
        else $finish$
        end if
    else $finish$
    end if
end procedure
```

Let us first introduce the patch as a simple pseudocode in Algorithm 1, since the patches themselves contain a lot of metadata and variables that could confuse the reader.

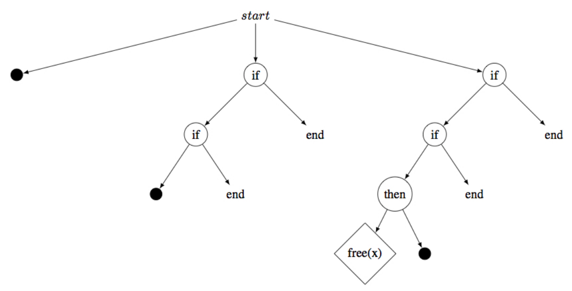


Figure 1: This figure illustrates the translation of the Patch explained with Algorithm 1 into an intermediate graph language

In the Algorithm 1, the pseudocode expresses a patch that has four different data modifications. The statements 2, 5, 12, and 13 are modifying some variables on the application. Once patched, the new state of the application will have four new values in some variables that have been updated from their previous state. However, if we carefully analyze this patch, we can see that there are only two variables being modified: *var* and *var*2. This patch, in the end, will have two variables modified according to the conditions implied by the branches.

The first subtree is a simple data declaration (i.e. write): this data modification is Machine 3 in Table 1. The second subtree is a combination of two if cases and a data modification, which is an expansion of Machine 5. This machine expresses an `if` case plus a data modification, but in our example we have two `if` cases instead. Finally, the third subtree consists of the same expansion of Machine 5 plus a data modification (Machine 1 in Table 1). The combination of all these machines defines a data patch.

3.4 Features and their relationship with graphs

By having a graphical representation of a patch, we are able to illustrate the control flow of the data modification statements. Figure 1 expresses the patch explained in Algorithm 1, and how data modification statements are defined on it. As it can be seen in Table 1, the leaves of each graph are the data modification statements (• and ⋄). These statements represent some modification of a variable or data structure, if we refer back to Figure 1 we can notice that the leaves are the • representing the use of an assignment operator (e.g. var = 0) or the ⋄ representing a function or operator from C modifying a variable (e.g. free(var)).

This is why the purpose of the graph representation language is to illustrate the control flow of data modifications, more precisely if we see an instance of a • know, based on the roots of the leaf, how to translate that statement into a data modification machine. As an example, if we refer to Figure 1, we can see that in order to modify data using the • on the leaves, we need to first compare different parameters with four different 'if cases'.

The purpose of the features of a graph is to characterize three different aspects that the translation tell us about the patches, that of the control flow of the patch, how data is being modify (e.g. which operators should we use over a database), and features related to the graph (e.g. number of subtrees, number of edges, cyclomatic complexity). Another feature is the **percentage of data operations**, which does not refer to how feasible the patch is to be implemented but how many of the statements of the patch are we able to express using basic database operations – some patches have a percentage of data operations of 40% and we are still able to patch them, as long as the statements that we are not able to express are not modifying data (i.e. not part of the 40% of the data operation statements).

Why did we choose these features?

For our experiments, we have three different type of features: graph representation, control flow and data operation features, we considered the latter two based on the approach of classifying patches by Stavrou et al. [34]. Recall that the *complexity of a patch* is the feature vector resulted by the translation process. For example, by having a feature vector we can measure its complexity by mapping it to its vector space. The following set of features were chosen according to a particular hypothesis about measuring the complexity of a patch.

3.5 Feature Vectors

We wanted to give a logical description for a patch that a machine could be able to understand. For this, we decided to focus on three aspects that we could define in our patches: control flow, data operations and our graphical representation of a patch. We deducted as many features as possible for each category and by the end we were able to construct a 15-Dimensional feature vector explaining these three categories. We think that most of the features are self-explanatory, or we could refer to Table 3.4 for more details, except for percentage of data operations which can be defined as the percentage of data operation statements in a patch (e.g. if we have a patch consisted of 10 statements, if 5 of them are related to data modifications then the percentage is of 50%) and the cyclomatic complexity feature, which is defined by the formula:

$$Cyclomatic\_Complexity = |E| - |V| + |N| \tag{1}$$

where E is the set of edges in the graph, V is the set of nodes, and N is the set of exit nodes.

We define the feature vector for our running example as:

$$\begin{aligned}
FV = [&20, & \text{Number of Operations,}\\
&16, & \text{Node Degree,}\\
&5, & \text{Longest Path,}\\
&8, & \text{Number of Searches,}\\
&4, & \text{Number of Compares,}\\
&3, & \text{Number of Writes,}\\
&1, & \text{Maximum input degree of a node,}\\
&2, & \text{Maximum output degree of a node,}\\
&0, & \text{Number of operations inside a loop,}\\
&0, & \text{Number of functions that return values,}\\
&1, & \text{Number of functions that pass parameters,}\\
&3, & \text{Number of subtrees,}\\
&15, & \text{Number of edges,}\\
&2, & \text{Cyclomatic Complexity,}\\
&100] & \text{Percentage of data operations in a patch]}
\end{aligned}$$

$$\overrightarrow{FV} = [20, 16, 5, 8, 4, 3, 1, 2, 0, 0, 1, 3, 15, 2, 100]$$

The resulting vector has a large number of features; linear classifiers are known to work better in a smaller dimensional space, thus we decided to reduce the dimensionality of the feature vectors. As detailed in the next section, we applied PCA over the set of features to obtain the most independent features from our set and to feed the vectors to our SVM.

4 Feature Vector Calibration

After undertaking a manual classification, graph and feature vector translation, and labelling of our dataset, we now have seventy five different input vectors for our SVM. However, before applying any machine learning algorithms, we must first standardize our dataset and reduce the dimension of our feature vectors. This reduction will help us to get better results when applying our SVM algorithm. For normalizing our dataset, we use a standard formula shown below. To reduce the dimensionality of our feature vectors, we use principal component analysis (PCA). Even though in SVMs, typically, the VC-Dimension of the feature space is much higher than the input space, we decided to reduce the dimensionality of the feature space using PCA to get a distribution of what are the most independent features in our dataset.

4.1 Standardizing Data

Each feature vector has many attributes that are correlated with each other. This is the best type of data to give as input to PCA. PCA gets possible correlated variables as input and maps it to a smaller dimensional subspace. In this way, it decides which features are the most independent and reduces our dimension to be less than or equal to the current number of features, in this case fourteen. However, before applying PCA we must first standardize the dataset. When we have values within the same range it is easier to see which features are really correlated with each other. Because the values are of mixed ranges, we decide to standardize the values of our dataset to be between {-1, +1} with the formula: $\frac{2(x)}{max(x)} - 1$

4.2 Applying PCA

We applied PCA using the free software programming language R [2]. We used the library "stats" and the function "princomp()" that takes as a parameter our data matrix.

We emphasize that PCA and SVM are not related or duplicate procedures here; we are purposefully using one (PCA) to impact the way the other (SVM) will work. We want to build an SVM that relies on the most independent features in the feature vector across all patches in the training set. Applying dimension reduction is a technique that helps us to get a better understanding of our data in terms of dependency, thus making it not only easier to get feature vectors in the future, but also to work with a smaller dimensional matrix which gives a smaller gramian matrix for our SVM.

According to the correlation plot in Figure 4, there is a large correlation between the features: number of writes, number of searches, computational

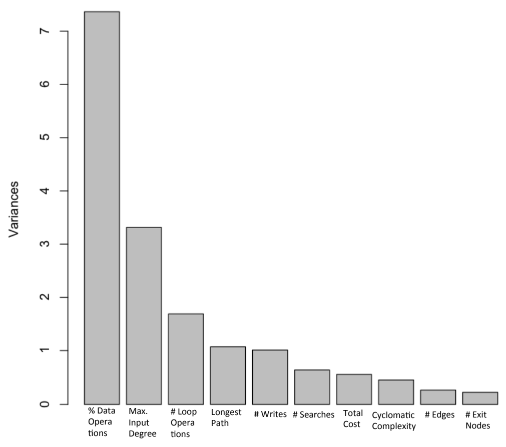


Figure 2: Variances of the features. According to Kaiser's criterion [39], we should only consider the components which variances are over 1, which are the first four features.

cost (i.e. number of primitive operations), cyclomatic complexity, number of edges and number of nodes. There is also some correlation between the number of compares and some other features related to relational algebraic operations used. According to Figures 4, 2, and 5 – and the definition of PCA – we should consider the least correlated variables: number of cyclic operations, maximum of input degrees, percentage of feasibility, and the longest path from the graph.

This means that, according to PCA and the correlation of our features, we can map the vector $\overrightarrow{FV}$ to $\overrightarrow{PCA\_FV}$ (data in this example is not standardized to make it easier to understand for the reader):

$$\overrightarrow{FV} = [20, 16, 5, 8, 4, 3, 1, 2, 0, 0, 1, 3, 15, 2, 100]$$

can be mapped to:

$$\begin{aligned} FV = [5, & \qquad \text{Longest Path,} \\ 1, & \qquad \text{Maximum input degree of a node,} \\ 0, & \qquad \text{Number of operations inside a loop,} \\ 100] & \qquad \text{Percentage of data modification statements]} \end{aligned}$$

4.3 Applying SVMs

There exist a wide variety of machine learning approaches, and it is not immediately apparent which technique is most suited to the task of classifying patches as a suitable or unsuitable candidate for hot patching. Classifying source code is not an easy task and there are many options (Neural Networks, Bayesian Classifiers, Genetic Algorithms, etc. [17]). However, we think that SVM is the best technique to use because many source code classification use SVMs by first translating the source code into an intermediate language.

SVMs are supervised learning techniques that analyze data and recognize patterns, this way classifying data according to our needs. It is a supervised learning technique because in order to get an output we first need to provide labeled data as input. Support Vectors, in SVM, are the data points closest to the hyperplane.

4.4 Experimental Methodology

We divided our data into training and testing sets to apply cross-validation. Our training dataset consists of 50 patches in their four-feature vector representations. Each vector was labeled with a Y/N according to a manual classification of feasibility. We manually went through each statement and determined if we were able to patch the data according to the statements. Recall that there is no functional relationship between a patch being infeasible and it being expressible in a relational algebraic language. The feasibility of a patch depends on if we are able to express all of its data modifications using our set of primitive operations.

We first train our machine using the 'ksvm' function in R and our training dataset. We used four different kernel functions – for the kernel trick in our SVM – to get different results when using our SVM. We used five different kernels: linear, polynomial, radial basis, anova and laplacian.

We train the SVM using R with the commands shown below. We first import data from CSV file and then train our machine using data_train (50 patches with labels Y/N).

```
> data_train <- read.csv("svm_train.csv", header = TRUE)
> data_test <- read.csv("svm_test.csv", header = TRUE)
> model_ksvm = ksvm(Label~., data_train, type = "C-svc",
kernel = "polydot", prob.model = TRUE)
```

We then use the `predict` function to classify our testing data, and we also classify our training dataset without the labels to see how well we are able to predict feasibility.

```
> predict(model_ksvm, data_test[,-5])
 [1] Y N Y N Y Y N Y Y N Y Y Y N N N N Y Y N N Y Y Y Y
Levels: N Y
> predict(model_ksvm, data_train[,-5])
 [1] Y Y Y Y Y Y N N N Y Y Y Y Y N
 N N Y N N Y Y N Y Y N Y N N Y Y N Y Y Y N Y N
[39] N N N N Y Y Y Y Y N N N
Levels: N Y
```

5 Evaluation

The results for our work are presented as a comparison between our manual classification of patches and our SVM results. Figure 3 compares our manually classified testing data with our SVM results. We are interested in answering the following questions:

5.1 Research Questions

1. How many patches can we express as a data patch? This question can be answered in the form of how many patches are defined as feasible. We manually go through our dataset and classify 75 different patches according to our definition of feasibility.
2. Are we able to train a machine to recognize a "feasible" patch? We reply to this question by feeding the 75 patches to the SVM and getting a set of labels for the patches as a result that define feasibility.
3. What is the relationship between the results we get when manually classifying a patch as feasible and the results given by our SVM? By getting the resulted set of labels we match them with the manual labels we used for our analysis of the dataset.
4. What defines a patch to be feasible according to our SVM? Finally, we reply this question by analyzing what the SVM defined as feasible and what do the patches that were labeled as feasible or infeasible have in common.

5.2 Experimental Methodology

We used a function in the R language to train and test our SVM [2]. After applying PCA over our data, we get our original space mapped to a 4-Dimensional space. We decided to choose the features resulted from our PCA analysis. One of the lessons from applying PCA is that the 4 features that were chosen as the most independent are each from a different feature category. We need to remind the reader that the three categories chosen for our features were: control flow, data operations and graph representation.

- **Longest Path of the graph:** There is a relationship between control flow and graph representation for this feature. The longest path of a graph is considered to be the number of nodes from the root to the leaves of the longest subtree of a graph. If we refer back to Figure 1, the longest path would be of 5 from the starting node to the Free() operator.

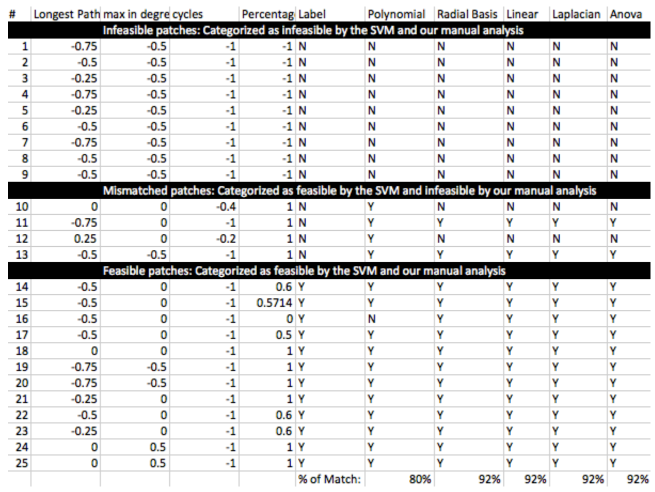
- **Maximum input degree of a node:** There is also a relationship between control flow and graph representation for this feature. This feature tells us what is the maximum of inputs of a node. The inputs are considered to be, for example, return values that modify a variable, variables that are modified inside a loop or a branch, etc. In Figure 1, the maximum input node degree is of 1, but if the value from a data modification within an if case is modified by the returning value of a function, then that node would have a degree of 2.

- **Percentage of data operations:** This feature is related to data operations of a patch. It tells, out of the statements of the patch, how many statements modify data structures? For example, if a patch has 10 statements and 3 of them modify data structures, then this features is 30%.

- **Loop Operations:** This feature is related to the control flow of the patch. It tells how many operations are within a loop (e.g. for case, while loop).

For the 4-Dimensional space, we use the library 'kernlib' in R to classify the dataset in a 4-dimensional plane. The 'kernlib' package offers us the option to provide a sigma value for the radial basis kernel function, which does a better job calibrating our SVM.

Why 50 training and 25 testing patches? There are many ways to analyze a set of data consisting of 75 different data points. We think that, the better we train our SVM, the better results we will get and the more the SVM will learn. This is why we decided to choose 2/3 of our dataset for training and 1/3 for testing purposes. For future work, we are planning to analyze a bigger set of patches and compare the results to what we learned from this experiment.



#	Longest Path	max in degre	cycles	Percentag	Label	Polynomial	Radial Basis	Linear	Laplacian	Anova
Infeasible patches: Categorized as infeasible by the SVM and our manual analysis										
1	-0.75	-0.5	-1	-1	N	N	N	N	N	N
2	-0.5	-0.5	-1	-1	N	N	N	N	N	N
3	-0.25	-0.5	-1	-1	N	N	N	N	N	N
4	-0.75	-0.5	-1	-1	N	N	N	N	N	N
5	-0.25	-0.5	-1	-1	N	N	N	N	N	N
6	-0.5	-0.5	-1	-1	N	N	N	N	N	N
7	-0.75	-0.5	-1	-1	N	N	N	N	N	N
8	-0.5	-0.5	-1	-1	N	N	N	N	N	N
9	-0.5	-0.5	-1	-1	N	N	N	N	N	N
Mismatched patches: Categorized as feasible by the SVM and infeasible by our manual analysis										
10	0	0	-0.4	1	N	Y	N	N	N	N
11	-0.75	0	-1	1	N	Y	Y	Y	Y	Y
12	0.25	0	-0.2	1	N	Y	N	N	N	N
13	-0.5	-0.5	-1	1	N	Y	Y	Y	Y	Y
Feasible patches: Categorized as feasible by the SVM and our manual analysis										
14	-0.5	0	-1	0.6	Y	Y	Y	Y	Y	Y
15	-0.5	0	-1	0.5714	Y	Y	Y	Y	Y	Y
16	-0.5	0	-1	0	Y	N	Y	Y	Y	Y
17	-0.5	0	-1	0.5	Y	Y	Y	Y	Y	Y
18	0	0	-1	1	Y	Y	Y	Y	Y	Y
19	-0.75	-0.5	-1	1	Y	Y	Y	Y	Y	Y
20	-0.75	-0.5	-1	1	Y	Y	Y	Y	Y	Y
21	-0.25	0	-1	1	Y	Y	Y	Y	Y	Y
22	-0.5	0	-1	0.6	Y	Y	Y	Y	Y	Y
23	-0.25	0	-1	0.6	Y	Y	Y	Y	Y	Y
24	0	0.5	-1	1	Y	Y	Y	Y	Y	Y
25	0	0.5	-1	1	Y	Y	Y	Y	Y	Y
					% of Match:	80%	92%	92%	92%	92%

Figure 3: Comparison between manually classification and predictions of our SVM for our testing dataset using every kernel function. The first set are the infeasible patches, then the mismatched patches that we considered to be infeasible, and the third set is the feasible patches.

5.3 Results

Answering the research questions that we posed above, we can conclude that we are able to express every data modification of 38 out of 75 patches. However, at least 24 out of the remaining 37 patches are uninteresting to us, because they are not creating new semantics on data structures.

Our dataset contains 75 patches that we classify into three categories: (1) patches that are infeasible because we cannot express their statements (e.g. they expect arbitrary computation such as user input), (2) patches that are infeasible, for our purposes, because they do not modify data structures (i.e. they have a 0% of data modification statements), or (3) patches that are considered to be feasible. We study how our SVM recognized these groups.

Experiment 1 To do so, we experimented with our dataset in two different ways. First, we divided our dataset into a training dataset (consisting of two–thirds of our data, 50 patches) and a testing dataset (with one–third of our data). We feed the training dataset to our SVM and predicted the testing dataset, and then we also use the training dataset as a separate testing dataset, without labels.

Experiment 2 Our second main experiment was an attempt to replicate the procedure and results of the first experiment by examining five randomly selected subsets of our dataset. This series of experiments randomly selected 30 patches as the training set and used the remaining 45 as our testing set. We predicted the labels for our testing dataset by using the same heuristics as before (that is, by using 5 different kernels).

The last two questions can be answered with our SVM results. Table 4, 3, and Table 5 contain our results. According to this data, SVMs appear to be a good way to predict feasibility of data patches. Using a polynomial kernel, our SVM predicted with 80% correctness the feasibility of our testing dataset (20 good results out of 25). Furthermore, it predicted with 84% correctness the feasibility of our training dataset (42 good results out of 50). This gives our SVM a 85% of correctness for our entire dataset. Using the other kernels (radial basis, linear, laplacian and anova) our SVM predicted with 92% correctness the testing dataset (23 out of 25 good results). On the training dataset, however, the best results were obtained using the radial basis and laplacian kernel functions (44 good results out of 50) with 88% of correctness. The results on Tables 3 and 4, tells us the best kernel functions to use with our dataset are radial basis and laplacian.

The results in Table 3, 4, and 5 tell us that the best kernel functions to use with our dataset are radial basis and laplacian. We emphasize that deciding feasibility of a patch is a non-deterministic task, therefore there is no algorithm or pattern that could tell us if a patch is feasible or not. However, by using our heuristics, we were able to classify patches using a machine learning algorithm.

5.4 Relationship between Feature Vectors and our Results

We thought that the best way to describe a patch was in the form of a feature vector that includes a representation of three important attributes that we considered for our patches: the control flow of the patch, the data operations, and our graphical representation, proposed in Section 3, for a patch. By the end we had a vector with 15 different features and, after applying PCA, we were able to map that vector into a 4-Dimensional space.

Now, the question we are trying to address on this subsection is: what is the relationship between these features and our SVM predictions? We can refer to Figure 5 and see that there is a notable relationship between the feature "percentage of data statements" and the label given by ourselves and the SVM. For example, for our testing dataset, we have 13 **infeasible** patches, on which only 4 had data modification operations – the rest did not have any data operations thus their data structures did not need to be updated. That leaves us with 12 **feasible** patches which all had over 50% of data operation statements, most of the 12 feasible patches had a bigger value for the "maximum input degree" feature than the infeasible patches; this means that the more complex the control flow is, the harder it is to hot patch data structures. And finally, there are no cyclic operations (i.e. for or while loops) in what the SVM classified as feasible patches.

The biggest confusion for our SVM was when a feature vector of a patch labelled as infeasible was the same as a feature vector labelled as feasible. For example, if we refer to Figure 3, we can see that Patch#11 has a similar feature vector as Patch#19 but different labels. This is because some of its data operations were involved in arbitrary computation, such as expected user input, and were not recognized by the SVM, thus mislabelling them as feasible.

5.5 Answering Research Questions

1. *How many patches can we express as a data patch?*: We are able to express 38 out of 75 patches as a data patch. From the remaining 37 patches, twenty-four patches are uninteresting to us because they do not modify the semantics of any data structure.
2. *Are we able to train a machine to recognize a "feasible" patch?*: The answer to this question is **yes**. By feeding the training and testing datasets to the SVM with no labels and was able to learn about the feature vectors and predict feasibility. We think that we can improve our results by isolating the patches that our SVM did not predict correctly and analyze them in order to get better features that could avoid confusion on our SVM. According to our results, the best kernels to use are Polynomial and Linear Kernels.
3. *How well does the machine learn to predict results according to the heuristics we use?*: Our SVM matched our labelling results with over 80% of correctness for our testing dataset and over 84% for out training dataset, which was treated as a new dataset by giving it to the SVM with no labels. For our second experimentation, we get an average correctness of 75.99% over our 5 datasets.
4. *What defines a patch to be feasible according to our SVM?*: From these results, we learned that there is a relationship between control flow and a patch's data structures being feasible to hot patch. The maximum input degree describes, for example, a statement using variables returned by calling a function. This makes hot patching data structures

infeasible because we cannot predict arbitrary computations that many functions do, we took a black-box approach when analyzing these functions. The cyclic operations are also related to the control flow of the patch, and the patches that were found to be feasible did not have any loops modifying data structures.

5.6 Limitations

There are a number of possible difficulties in applying a patch or sources of error in this type of work, including:

1. we can't express patch semantics in our "language"
2. the patch may actually be malicious
3. the patch may be broken
4. the patch may be complex and difficult to categorize / classify
5. the patch may contain certain elements that have an unpredictable impact on data semantics (e.g., input, unbounded computation)
6. the ML just was not powerful enough to make a correct decision (classifier wasn't trained well enough / sensitive / or calibration enough)
7. manual error / human error in manual classification / assessment of patch characteristics.

6 Related Work

Applying machine learning techniques to the problem of patching is, to the best of our knowledge, something new to the machine learning field, even if machine learning has been applied in the security realm before (especially intrusion detection). However, classifying source code with a signature-based approach is similar to what we are trying to do in our work. This has been done in applying machine learning techniques on virus analysis [15, 28] and similarly with spam analysis [37], where spam is classified according to a pre-determined set of key words.

Software self-healing has been an active area of research. In one early example, *failure oblivious computing* executes through faults [30]. The Reactive Immune System [31] aims at roughly the same concept: process execution can be forced through a fault or exploited vulnerability by "slicing off" the corrupted function and returning an error code. Instead of attempting to force continued execution through an exploited vulnerability, a significant some work attempts to rewind execution to a pre-fault or otherwise uncorrupted state [4, 11, 16, 26].

As the community explores the promise of more ambitious software repair efforts [21, 26, 30–32, 38] or hot-patching OS kernels [3, 7]) to automatically defend systems from new attacks and vulnerabilities, we must recognize that such techniques carry a great deal of risk because they largely bypass the cycle of patch testing used to vet both vendor and internally developed patches. Such techniques would benefit from an automated method of classifying a patch or fix, and this paper details such a procedure and approach.

7 Conclusion

We demonstrate a method for predicting whether a security patch is suitable for "hot patching" (i.e., an unsupervised dynamic software update to ameliorate a vulnerability). We train an SVM to identify patches that contain constructs that would make it difficult to automatically apply the patch to a running application. The feature vectors of the SVM represent the "complexity" of the patch in terms of its operations on data and control flow.

The SVM classifies 92% of our training set and 84% of our testing set as *feasible*; this rate means that 4 out of 5 times, our SVM can help a system owner decide whether a patch is a good candidate for applying to their running system as a dynamic software update (with concommitant "live" data updates). We also learned that the control flow of a patch (i.e. loop operations, returned values by functions, number of nested operations in a branch or loop) are elements that define feasibility of a patch in terms of how they modify data structures.

References

[1] `http://fetchmail.berlios.de/fetchmail-SA-2005-01.txt`.

[2] David Meyer Alexandros Karatzoglou. Support vector machines in r. In *Journal of Statistical Software*, 2006.

[3] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots Are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly. In *Proceedings of the USENIX Annual Technical Conference*, June 2007.

[4] A. Brown and D. A. Patterson. Rewind, Repair, Replay: Three R's to dependability. In 10th *ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[5] Aaron Brown, Daniel Hettena, Jon Kuroda, Noah Treuhaft, David A. Patterson, and Katherine Yelick. Roc-1: Hardware support for recovery-oriented computing. *IEEE Trans. Comput*, 51:2002, 2002.

[6] George Candea and Armando Fox. Crash-Only Software. In *Proceedings of the* 9th *Workshop on Hot Topics in Operating Systems (HOTOS-IX)*, May 2003.

[7] Silvio Cesare. Runtime Kernel kmem Patching, 1998. `http://vx.netlux.org/lib/vsc07.html`.

[8] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 271–281, Washington, DC, USA, 2007. IEEE Computer Society.

[9] Dustin Childs. Kb2839011 released to address security bulletin update issue.

[10] A. Di Stefano, G. Pappalardo, and E. Tramontana. An infrastructure for runtime evolution of software systems. In *Computers and Communications, 2004. Proceedings. ISCC 2004. Ninth International Symposium on*, volume 2, pages 1129–1135 Vol.2, 2004.

[11] G. W. Dunlap, S. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, February 2002.

[12] Brian Fahs. Spedi: Static patch extraction and dynamic insertion. *IEEE Trans. Comput*, 51:2002, 2002.

[13] Daniel Fleshbourne. iphone users now fear security patches, say analysts.

[14] Stefan Frei, Thomas Duebendorfer, and Bernhard Plattner. Firefox (in) security update dynamics exposed. *SIGCOMM Comput. Commun. Rev.*, 39(1):16–22, 2009.

[15] D. Gavrilut, M. Cimpoesu, D. Anton, and L. Ciortuz. Malware detection using machine learning. In *Computer Science and Information Technology, 2009. IMCSIT '09. International Multiconference on*, pages 735–741, 2009.

[16] Samuel T. King and Peter M. Chen. Backtracking Intrusions. In 19th *ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[17] J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2006, 2006.

[18] Evgenios Konstantinou. Metamorphic virus: Analysis and detection, 2003.

[19] F. Maggi, W. Robertson, C. Kruegel, and G. Vigna. Protecting a Moving Target: Addressing Web Application Concept Drift. In *Proceeding of the* 12th *International Symposium On Recent Advances In Intrusion Detection*, 2009.

[20] Elinor Mills. Symantec Pulls Norton Patch After Error Reports. *CNET News: Insecurity Complex*, August 2009.

[21] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Commun. ACM*, 51(12):87–95, 2008.

[22] Nilay Patel. Botched McAfee Update Shutting Down Corporate XP Machines Worldwide. In *http://www.engadget.com/2010/04/21/mcafee-update--shutting-down-xp-machines/*, 2010.

[23] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[24] P.Li, D. Gao, and M. Reiter. Automatically Adapting a Trained Anomaly Detector to Software Patches. In *Proceeding of the* 12th *International Symposium On Recent Advances In Intrusion Detection*, 2009.

[25] Bogdan Popa. Windows 8 update fails on kb2770917.

[26] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*, 2005.

[27] Ashwin Ramaswamy, Sergey Bratus, Michael E. Locasto, and Sean W. Smith. Katana: A Hot Patching Framework for ELF Executables. In *The 4th International Workshop on Secure Software Engineering (SecSE 2010), held in conjunction with ARES 2010*, February 2010.

[28] Christopher Richardson. Virus detection with machine learning, 2009.

[29] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 108–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and Jr. W Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[31] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 149–161, April 2005.

[32] A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *Proceedings of the 12th Symposium on Network and Distributed System Security (NDSS)*, February 2005.

[33] Craig A. N. Soules, Jonathan Appavoo, Dilma Da Silva, Marc Auslander, Gregory R. Ganger, Michal Ostrowski, and et al. System support for online reconfiguration, 2003.

[34] Angelos Stavrou, Gabriela F. Cretu-Ciocarlie, Michael E. Locasto, and Salvatore J. Stolfo. Keep Your Friends Close: The Necessity for Updating an Anomaly Sensor with Legitimate Environment Changes. In *Proceedings of the 2nd Workshop on Artificial Intelligence and Security*, 2009.

[35] Thomas Stibor. A study of detecting computer viruses in real-infected files in the n-gram representation with machine learning methods. In *Proceedings of the 23rd international conference on Industrial engineering and other applications of applied intelligent systems - Volume Part I*, IEA/AIE'10, pages 509–519, Berlin, Heidelberg, 2010. Springer-Verlag.

[36] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: a vm-centric approach. *SIGPLAN Not.*, 44(6):1–12, June 2009.

[37] Konstantin Tretyakov. Machine learning techniques in spam filtering. Technical report, Institute of Computer Science, University of Tartu, 2004.

[38] Westley Weimer, ThanVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically Finding Patches Using Genetic Programming. In *International Conference on Software Engineering (ICSE)*, 2009.

[39] Keith A. Yeomans and Paul A. Golder. The guttman-kaiser criterion as a predictor of the number of common factors. In *In The Statistician Vol. 31, No. 3*, 1982.

Appendix

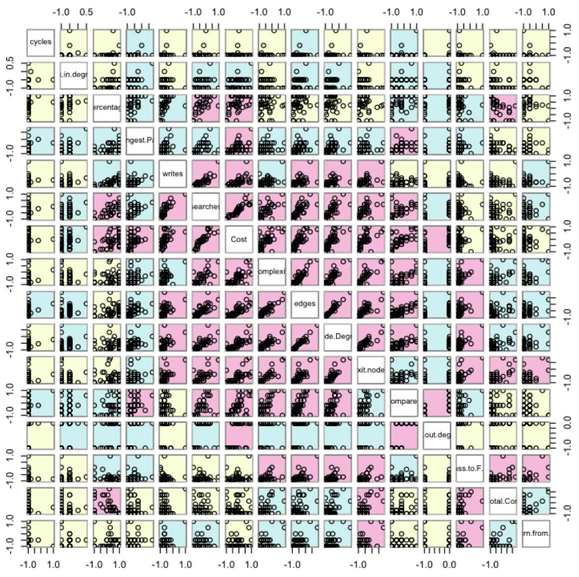


Figure 4: This figure illustrates the correlation of all the features. The darker the colors are, the more correlation there is. In this case we care for the lighter squares to look for less dependency, this is why we chose the first four features (matching with the results in Figure 2)

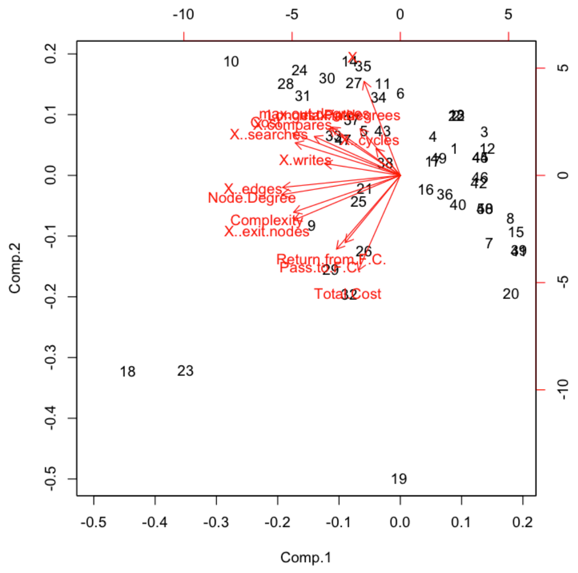


Figure 5: The figure shows the correlation of all the features as vectors. The closer the vectors are, the more correlated.

Table 1: Data Modifications Machines

Machine	Name	Format	Example	Operations	Computational Cost	Node Label
1	Simple Data Mod.	var_name = value	data = 10;	**search**(data) and **write**(10) into data	2	$\bullet$
2	Simple Data Mod.	var_name = var2_name	data = data2;	**search**(data), **search**(data2) in local_vars and **write**(data2) into data	3	$\odot \rightarrow \bullet$
3	Declaration New Var.	type var_name;	int data;	**write**(int, data) into local_vars	1	$\odot$
4	Declaration New Var.	type var_name = value;	int data = 10;	**write**(int, data, 10) into local_vars	1	$\odot$
5	If case + Data mod.	if (var < value) then var = value;	if(data < 10) then data2 = 0;	**search** (data) *in local_vars*, **read**(data), **compare**(data<10), **search**(data2), **write**(0) *into data2*;	5	$\diamond^1 \leftarrow \bigcirc \rightarrow \bullet$
6	If case + Else case + Data mod.	if (var < value) then var = value; else var = diff_value;	if(data < 10) then data2 = 0; else data2 = 10	**search** data *in local_vars*, **read**(data), **compare**(data<10), if it is then **search**(data2), **write**(0) *into data2*; else **search**(data2), **write**(10) *into data2*;	5	$\bullet \leftarrow \bigcirc \rightarrow \bullet$
7	Data Modifications using Operators and Functions using metadata	sizeof(var) free(var)	if (sizeof(var) ¿ 4) then free(var);	**search**(var.size), **read** (var.size), **compare**(var.size ¿ 4), **search**(NULL) into *var*	Depends on the operator	$\diamond$

Feature Type	Features	Explanation
Control Flow	Number of Operations # Loop Operations # Functions that Return Variables # Functions that Pass Parameters	We chose these features because one of our hypothesis is that the control flow of a patch define its complexity.
Data Operations	# Searches # Compares # Writes % Data operations out of statements	Another hypothesis is that the amount of data modification statements, on the patch, define how hard it is to hot patch it.
Graph Representation	Node Degree Longest Path Max. In Degree Max. Out Degree # Subtrees # Edges	The third hypothesis is that the complexity of the graph defines the complexity of a patch.

Table 2: Different types of features of our vector

Kernel Function	% Correct Classification	Results
Polynomial	84%	42 out of 50
Radial Basis	88%	44 out of 50
Linear	84%	42 out of 50
Laplacian	88%	44 out of 50
Anova	84%	42 out of 50

Table 3: *Results for Training Data Set for Experiment #1*

Kernel Function	% Correct Classification	Results
Polynomial	80%	20 out of 25
Radial Basis	92%	23 out of 25
Linear	92%	23 out of 25
Laplacian	92%	23 out of 25
Anova	92%	23 out of 25

Table 4: *Results for Testing Data Set for Experiment #1*

Random Dataset	Polynomial	Radial Basis	Linear	Laplacian	Anova
First Random Set	66.66%	66.66%	66.66%	66.66%	71.11%
Second Random Set	80%	75.55%	80%	75.55%	73.33%
Third Random Set	80%	75.55%	80%	77.77%	75.55%
Fourth Random Set	80%	75.55%	80%	77.77%	77.77%
Fifth Random Set	80%	74.66%	77.33%	74.66%	75.99%
Averages	77.33%	74.66%	77.33%	74.66%	75.99%

Table 5: *Results for Testing Data Set for Experiment #2*