2014-02-05

DPL: A Data Patching Language

Gonzalez, Robin

UNIVERSITY OF CALGARY

DPL: A Data Patching Language

by

Robin Gonzalez

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTERS OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JANUARY, 2014

Abstract

Patching applications remains one of the most effective techniques for defending against exploitation of vulnerabilities and is a basic defensive mechanisms against attacks. However, it entails unwanted complications for the user, such as restarting the application after it gets patched. Restarting the application influences the user to stop updating applications and operating systems, making out of date software that presents an attractive target for exploitation. Even though many authors address this issue by proposing frameworks and tools for applying these patches *on the fly*, most modern systems and applications do not implement this technique. One of the biggest challenges for mainstreaming this technique is the fact that patches not only change source code but also the state or semantics of the application.

This thesis proposes a mechanism that aids the activity of hot patching applications by updating its data semantics while dynamically applying a patch. More precisely, the mechanism updates the data structures of an application according to what a patch entails by making sure that the application's state is updated according to the new semantics introduced by the patch. For this, we present a proof of concept of a framework that is capable of patching the data semantics (i.e., data structure modifications according to a security patch) of an application.

This thesis explores the question of what makes a patch feasible for hot–patching according to how it modifies the semantics of an application. We also study the application of machine learning algorithms to predict patches that were considered to be feasible for hot–patching based on an empirical study. We also explain the design and implementation of a proof of concept capable of hot–patching data structures of applications.

As in many other scientific studies, we found that there is a subset of patches that we are not able to use for hot–patching because of the operations they are introducing. By

studying this subset of patches, we learned that certain data operations introduce changes in the control flow that can create conflicts when hot–patching. We explain what type of operations defined a patch to be infeasible – according to our heuristics – and we hot–patched the statements that we found to be feasible. Our system is capable of hot–patching different types of data structures according to the aforementioned feasible operations with a very low performance overhead.

At the end, we present the evaluation and results of our investigation. We learned that 13 out of 75 security patches that modify data structures are not feasible to implement using our heuristics, making them difficult to update because of the semantics the patch introduces. On the other hand, we found 38 out of 75 security patches feasible to implement by using our set of data operations and the remaining 24 were not modifying data semantics. In conclusion, we found that, if patch developers are aware of the type of statements that introduce conflicts when hot–patching, they could make hot–patching a feasible activity.

Acknowledgements

The more I see, the less I know for sure. - John Lennon

The "well–informed" think they know something about matters that the experts are reluctant even to speak of. Information at second hand always gives an impression of tidiness, in contrast with the data at scientist's disposal, full of gaps and uncertainties. - Stanislaw Lem

The last two years have been quite a journey for me and all the people that I love. Moving to a new country, and living in a new and very different culture from Venezuela, has changed my life in many ways, influencing on my definition of **happiness**. I feel really happy and grateful with my life and the opportunities that God has given to me. I am very grateful for having the opportunity of writing a master's thesis and furthering my educational background, specially in a university such as the University of Calgary.

I would like to thank Dr. Michael E. Locasto, my supervisor, whose patience and knowledge have been the key to learning and wanting to be more involved in the security field and becoming a Master in Computer Science. Thanks for believing in me and giving me one of the best opportunities of my life; I will always appreciate that. I would also like to thank one of my biggest influences in my religious perspectives and educational purposes: "Brother" Antonio Eguia, I hope you are resting in peace up in Heaven, we will always miss you!

I would also like to thank all of my family members: cousins, uncles, aunts, and grandparents; specially to my *mami*: Ana Teresa, which dedication and hard work has always given me strength and the desire to become a better person; my dad: Robinson, that has always supported me in every decision that I have made in my life and my sisters Vanessa and Stephanie who are the biggest joys of my life. I would also like to thank Abuela Estela, Abuelo Enrique, Abuelo Robinson and Abuela Alicia for always being there for me, praying

for my success. Special thanks to Ivan and Luisa for helping me get on my feet. I love you all.

I feel very grateful to all the friends I have made here in Calgary and in my early life in Venezuela, specially in my childhood while studying in Los Maristas school. I would like to thank (in alphabetical order): Becky, Diego, Eduardo, Ender, Felix, Gino, Jess, Jon, Josbel, Jose, Oswaldo, Sarah, Sutapa, Travis and many others who always supported me during hard times in Calgary.

Last, but not least, I would like to thank Daniel, Jonathan, Sarah and those who read and criticized my work, helping to make it better and more *understandable*. And of course, I would like to thank God for giving me strength and desire to always overcome any difficulties. The best lesson I have learned in life is that **God is happiness**. Last, but not least, I would like to thank *you*, the reader, for making some time to read my thesis. I hope you enjoy it.

When the world that surrounds you is aware of your virtues, that is when you started to live. - Robin Gonzalez

Table of Contents

Abstract . . . i
Acknowledgements . . . iii
Table of Contents . . . v
List of Tables . . . viii
List of Figures . . . ix
List of Symbols . . . xi
1 Introduction . . . 1
1.1 Limitations of Patching . . . 2
1.2 Learning to Data Patch . . . 3
1.2.1 Key Challenge: Updating Data Structures . . . 4
1.3 Thesis Scope . . . 7
1.4 Motivation . . . 9
1.5 Thesis Statement . . . 10
1.5.1 Hypotheses . . . 10
1.6 General Research Questions . . . 11
1.7 Contributions . . . 11
1.8 Thesis Overview . . . 13
2 Background and Related Work . . . 15
2.1 Security Patches . . . 15
2.1.1 Common Vulnerabilities and Exposures . . . 16
2.1.2 Limitations of Security Patches . . . 19
2.2 Components of DPL as a System . . . 19
2.2.1 DWARF information . . . 20
2.2.2 Pin: Dynamic binary instrumentation . . . 20
2.3 Related Work . . . 20
2.3.1 Self–healing mechanisms and Langsec . . . 21
2.3.2 Hot Patching and DSU . . . 22
2.3.3 Genetic Programming . . . 24
2.3.4 Data Structure Modifications and Shape Analysis . . . 24
2.3.5 Machine Learning Techniques on Source Code . . . 26
2.4 Unaddressed Challenges . . . 26
3 Design Overview . . . 28
3.1 Workflow and Experiments . . . 28
3.2 Manual Analysis of Patches . . . 31
3.2.1 Translation of Source Code into Graph Language . . . 32
3.2.2 Features of a Patch . . . 33
3.2.3 Feasible and Infeasible . . . 34
3.2.4 Labeling . . . 35
3.3 **Applying Machine Learning on Patches** . . . 35
3.3.1 Applying PCA over data . . . 37
3.3.2 Using Support Vector Machines . . . 37
3.4 DPL Tool . . . 38

3.4.1 Pin Library . . . 38
3.4.2 Data Updating . . . 39
3.4.3 Data Patch Object . . . 40
4 Analysis of Patches . . . 42
4.1 Intermediate Language for Machine Learning . . . 42
4.2 Our data–patching language . . . 43
4.3 Feasible and Infeasible Patches . . . 44
4.4 Data Modification Machines . . . 45
4.5 Constructing Graphs From Patches . . . 47
4.5.1 Features and their relationship with graphs . . . 50
4.5.2 Feature Vectors . . . 51
4.6 Summary . . . 53
5 Applying SVMs . . . 56
5.1 Feature Vector Calibration . . . 56
5.1.1 Standardizing Data . . . 57
5.1.2 Applying PCA . . . 57
5.1.3 Experimental Methodology . . . 58
5.2 Summary . . . 60
6 DPL: A Data Patching Language System . . . 64
6.1 Design Overview . . . 64
6.1.1 Import Routine . . . 65
6.1.2 Patching data structures . . . 65
6.1.3 Export Routine . . . 67
6.2 Storing data structures according to their type . . . 68
6.2.1 Primitive types . . . 69
6.2.2 More complex types of structures . . . 73
6.3 Data Patch Object . . . 75
6.4 Implementation . . . 79
6.4.1 Pin: A Dynamic Binary Instrumentation Tool . . . 79
6.4.2 DWARF info . . . 83
6.4.3 Patching the source code . . . 85
6.4.4 Summary of DPL System . . . 86
6.5 Applications for DPL . . . 87
6.6 Limitations of DPL . . . 87
7 Evaluation Methodology . . . 89
7.1 Evaluating our Empirical Study . . . 89
7.1.1 Analysis of Patches . . . 90
7.1.2 Applying SVMs for Predicting Feasibility . . . 92
7.2 Using DMMs for Security Patches . . . 93
7.2.1 Patching our Data Modification Machines . . . 94
7.2.2 Using DMMs to Update Data Semantics According to Security Patches 97
7.3 Patching Data Structures of Applications with DPL . . . 99
7.3.1 Patching C Applications . . . 99
7.3.2 Patching More Complex Data Structures . . . 100
7.3.3 Analysis for Patching More Complex Applications . . . 103

7.4 Research Questions 105
8 Results 106
8.1 Analysis of Patches 106
8.1.1 Relationship between Feature Vectors and our Results 109
8.2 Discussion of the DPL System 111
8.2.1 Revisiting our data modification machines (DMMs) 112
8.2.2 Patching real-life security patch models 115
8.2.3 Modifying data structures with DPL 116
8.2.4 Performance Evaluation 117
8.3 Wrapping it up: Answering our Research Questions 118
8.4 Discussion 122
8.5 Future Work 124
Conclusions 125
Bibliography 127

List of Tables

4.1 Description of data modification machines. 54
4.2 Different types of features for our feature vector. 55

7.1 How to use DMM#1 in C and SQL. 94
7.2 How to use DMM#2 in C and SQL. 95
7.3 How to use DMM#3 in C and SQL. 95
7.4 How to use DMM#3 in C and SQL. 96
7.5 How to use DMM#4 in C and SQL. 96
7.6 How to use DMM#4 in C and SQL. 97
7.7 How to use DMM#5 in C and SQL. 97
7.8 How to use DMM#6 in C and SQL. 98
7.9 Security Patches used to demonstrate the capabilities of DPL as a hot–patching system 98

8.1 Results for the classification of our training data set with no labels. 106
8.2 Results for Testing Data Set using our first experiment (1/3 of dataset). . . . 108
8.3 Results for testing data set using our second experiment (3/5 of dataset). . . 109
8.4 DMMs that we can patch with DPL. 113
8.5 Hot–patching data structures of applications using security patches. 116
8.6 Our test suite for hot–patching data structures with DPL. 119

List of Figures and Illustrations

1.1 Simple patch illustrating if case + data modification. 4
1.2 First security patch example for Samba. 6
1.3 Second security patch example for Samba 7
1.4 DPL patching mechanism diagram. 8

2.1 Example of isolation of operations in a security patch. 17
2.2 Example of a CVE entry. 18

3.1 Diagram for the process of analyzing and classifying patches. 36

4.1 Graph representation of the control flow of data modifications for Algorithm 1. 48
4.2 Graph representation of the control flow of data modifications for Algorithm 2. 49

5.1 Plot of the correlation of features using PCA. 62
5.2 Plot of the correlation of features using Kraiser's criterion. 63
5.3 Correlation of features using vectors. 63

6.1 Variables we update according to scope. 66
6.2 Example of global data structures in C. 69
6.3 Source code for returning address with DWARF. 71
6.4 Example of a Debugging Information Entry (DIE) in DWARF. 71
6.5 Source code for dereferencing addresses in DPL. 72
6.6 Example of local variables in C. 72
6.7 List for storing local variables in DPL. 72
6.8 Getting the DIE of a structure with DWARF. 75
6.9 Detecting complex structures using DWARF. 76
6.10 Example of dynamically allocated data structure in C. 76
6.11 Data patch object 77
6.12 An example of a security patch for HTTPD using only one of our DMMs. . . 79
6.13 Diagram for the hot–patching of data structures process using DPL. 80
6.14 Getting *ebp, eip,* and *esp* using DPL as a Pin tool. 81
6.15 Main function in the source code of DPL. 82
6.16 Analyzing the context of a CPU architecture using DPL as a Pin tool. 82
6.17 Example of a set of DIE frames inside the DWARF information. 84
6.18 Getting the size of an array using DWARF information. 86

7.1 Primitive types of data structures we update. 100
7.2 Source code used for patching a hello world program using the DPL system. 100
7.3 Output of hello program after using DPL. 101
7.4 Data patch object for patching the hello world program. 101
7.5 Example of table in our database after importing a global data structure. . . 101
7.6 Example of a table in our database after importing a linked list with DPL. . 102
7.7 An example of a linked list implementation that was correctly patched by the DPL System. 103

7.8 A four–step procedure for hot–patching applications with DPL. 104

8.1 Testing dataset results when classifying with SVM. 107
8.2 Radial correlation plot of the longest path versus maximum of an input degree node. 108
8.3 Radial correlation plot of the maximum of an input degree node versus number of cyclic operations. 109
8.4 Radial correlation plot of the longest patch versus percentage of data operations. 110
8.5 Linear correlation plot of the longest path versus maximum of an input degree node 111
8.6 Linear correlation plot of the maximum of an input degree node versus number of cyclic operations. 112
8.7 Linear correlation plot of the longest patch versus percentage of data operations. 113
8.8 Heat map representation of feature vectors with leave–one–out approach. . . 114

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
API	Application Programming Interface
AST	Abstract Syntax Tree
CVE	Common Vulnerabilities and Exposures
DB	Database
DIE	Debugging Info Entries
DMM	Data Modification Machine
DPL	Data Patch Language
DSR	Data Structure Recovery
DSU	Dynamic Software Updating
DWARF	Debugging With Attributed Record Formats
ELF	Executable and Linkable Format
OS	Operating System
PCA	Principal Component Analysis
PID	Process ID
SDM	Simple Data Modification
SVM	Support Vector Machine

Chapter 1

Introduction

Patching remains one of the best practices for securing our computer systems. However, many users and organizations tend to avoid it because it implies restarting their systems or applications, and thus making it a tedious, slow, and inconvenient activity. Restarting an application after updating it has remained an issue since the beginning of modern computer systems, to the point where the Defense Advanced Research Agency (DARPA) offers two million dollars for the development of an automated network defense system that patches vulnerabilities on the fly [18]. But, why do we need to patch applications in the first place?

No software is perfect. Most software has bugs, therefore most software has vulnerabilities. Attackers take advantage of these vulnerabilities [61], and often use them to exploit users running those vulnerable applications. Both vendors and an active security community constantly look for these vulnerabilities and fix them by modifying the vulnerable part of the source code. A detailed report is then sent to the developer of the application including a description of the vulnerability and a fix in the form of a **patch**.

By analyzing related work and bugs, errors, and failures that patches caused, we believe that one of the biggest obstacles for hot patching (i.e., updating without restarting the application) is the modification of memory–resident data structures. The central question we address in this thesis is: can hot-patching data structures become feasible through the design of a principled mechanism to update data structures according to a patch? To answer this question, we divided this thesis into three main research focus areas: analysis of patches, machine learning on patches, and DPL as a real, operational software patching framework.

To understand how patches modify data structures, we first analyzed 75 security patches of different open-source projects, each patch having unique characteristics. An empirical

study of these patches helps us understand how these patches modify applications (more specifically, the application's data structures).

We then study a mechanism that, given a patch, applies machine learning algorithms to predict if the framework is capable of hot–patching the application according to how the patch modifies data structures. We finish this thesis by presenting an operational proof of concept for a framework that patches the data structures of applications according to security code patches that were predicted as feasible by our machine learning algorithm.

1.1 Limitations of Patching

Patching and keeping applications up to date remains one of the best defenses to secure computer systems. However, users tend to avoid patching applications because of the need of restarting the system after updating.

The question is, even though many authors [29, 60, 46, 49] are addressing hot–patching by introducing many different frameworks, why does hot–patching remain a research topic rather than a commonly deployed system capability? Patching an application is a hard task that may cause conflicts, and most developers do not study all of the potential conflicts when developing a patch. Therefore, it is difficult to predict the behaviour of a patched application, when we do not know the semantics the patch entails. Patches sometimes bring unexpected behaviour to applications [1]; many patches cause conflicts once applied, resulting in a cascading series of patches [1, 14, 21, 47]. In this thesis we study how to address the conflict of patching data structures on an application within a live process address space.

One of the main challenges of hot–patching a program is updating the context or semantics that the source code of the patch modifies. The purpose of restarting or rebooting the system is to restore the state of the program, this is, however, a disruptive activity that interrupts the workflow of end–users. There are many applications (e.g., web–servers)

[1] A Norton patch [41] and a McAfee update [45] both presented unexpected behaviour which required users to patch the already patched application. These cases are called cascade–patching.

that cannot afford being restarted. For these applications, availability is important. We are studying in this thesis how to recover and update these semantics, more specifically the ones related to data structures. For doing this we randomly sampled 75 security patches and, after an empirical evaluation of this dataset, we discovered that we are able to update 50.7% of the dataset, we are not able to update 17.3%, and the remaining 32% are not modifying any data structure and thus not modifying the data semantics of the program.

1.2 Learning to Data Patch

When compared to software, patches are simpler to analyze and reverse-engineer because they are a subset of a larger set of statements representing the software. However, they also rise some challenges, such as deduction of the context of the patch's statements, that we need to understand in order to achieve a successful live–update.

One of the most difficult challenges of hot-patching an application is understanding the *context* or *semantics* of the statements we want to patch. Patches operate on a group of statements that need to be updated because of vulnerabilities they bring to the application. However, they do not specify the new semantics that the patches bring to the application.

For the purpose of gaining a broad understanding of how patches modify the operational semantics of existing data structures, we analyze a set of seventy-five patches from a variety of open-source projects. After analyzing the set of patches, we were able to conclude that the statements of the patch that modify data structures can be translated to a set of primitive operations similar to those encountered in database operations. This is related to one of our hypotheses in Section 1.5.1, which states that these primitive operations are regular enough to model in a principled fashion.

Based on this observation, we define this set of operations as $read, write, compare, search$ and one of the objectives of this thesis is to understand how we can translate statements encountered in patches into these operations, and then dynamically apply them using our

```
+  if (list[i] > 10)
+      list[i] = 0;
```

Figure 1.1: An example of a patch for the algorithm ApplicationExample that modifies the condition from "elements need to be below 10" to "elements need to be above 10". The + statements in the patch are the ones we are introducing to the application.We focus on patching the new data semantics security patches are introducing and this is one type of operation we found in our dataset analysis. We defined the operation as an if–case modifying a data structure.

system presented in Chapter 6.

1.2.1 Key Challenge: Updating Data Structures

One of the techniques to dynamically apply a security patch into an application is called Dynamic Software Updating (DSU). DSU has been extensively studied in the past by several authors [60, 49, 46, 44, 19, 56, 13]. However, it has not been completely achieved or implemented in current systems. Most commonly used operating systems and applications need to be restarted after updating. In this thesis, we address the issue of updating data structures in the application according to a security patch. Imagine a very simple application, such as one that updates elements in a list according to their values, as an example refer to the Algorithm proposed bellow.

```
1: procedure ApplicationExample
2:     int list [10] // statement representing a list of elements
3:     for (i = 0; i<10; i++) // lookup elements in the list
4:     if (list[i] < 10)
5:     list[i] = 0; // make the element in the list equal to 0 if the value is less than 10
6: end procedure
```

Let us suppose that we found a vulnerability related to elements with values below 10. We want to apply a patch that modifies lines 4 and 5 of the application's source code. For that, we can issue a patch that modifies the source code as in Figure 1.1.

We are not only patching an arbitrary piece of source code, but also changing the semantics of some data structure in the program. The patch can also be read as "from now on, whenever *list[i]* is bigger than 10, make *list[i]* equal to 0." This example help us start analyzing how patches modify data structures. The critical research question we focus in this thesis is an examination of these modified semantics and whether they can be automatically applied to existing data in the program's memory space.

In the example, we need to meet a *condition* (the element to be compared to 10) in order to modify $list[i]$, which is an element of a list. Previous hot patching frameworks would focus on updating the source code of the application, but there is something more happening in context: the modification of data structures. If we only update the patch code of the program, the list will remain with its previous states or data values. We want *list* to be updated according to what the patch *implies* with its new data semantics.

We want to **data patch** *list* and update it according to the new semantics the patch in Figure 1.1 entails. More precisely, to analyze the data semantics of *list* we **compare** the $list[i]$ element to the value 10 then, if the case is valid (i.e., if the value is 10), **modify** the value of *list[i]* to be equal to 0, otherwise leave it as is. This combination of operations is what we define in this thesis as a **data patch**.

Our purpose is to add a feature to security patches that makes hot–patching easier to achieve. Since we focus on hot–patching data structures, we designed a relational–algebraic language capable of updating these data operations by using our four operators search, read, write, and compare. One attribute that was challenging to retrieve was the address of memory–allocated data structures, we achieved this by using Pin [37], and the DWARF info [66]. We think it would be ideal (i.e., making hot–patching a feasible activity) to implement a mechanism capable of tracking the type of memory allocation results in a programming language like C. A data patch for Figure 1.1 constructed using our language would be as follows:

1: **procedure** PRIMITIVE OPERATIONS TRANSLATION OF PATCH1
2: **search**: list
3: **read**: list[i]
4: **compare**: list[i] > 10
5: if (list[i] > 10) then: **write**(0) **into** list[i]
6: else: do nothing
7: **end procedure**

```
@@ -1160,7 +1160,7 @@ int chain_reply(char *inbuf,char *outbuf,int size,int bufsize)
        static char *orig_inbuf;
        static char *orig_outbuf;
        int smb_com1, smb_com2 = CVAL(inbuf,smb_vwv0);
-       unsigned smb_off2 = SVAL(inbuf,smb_vwv1);
+       static unsigned smb_off2;
        char *inbuf2, *outbuf2;
        int outsize2;
        int new_size;
@@ -1178,8 +1178,16 @@ int chain_reply(char *inbuf,char *outbuf,int size,int bufsize)
                /* this is the first part of the chain */
                orig_inbuf = inbuf;
                orig_outbuf = outbuf;
+               smb_off2 = 0;
        }
```

Figure 1.2: Example of a source code patch. The - sign represents the old statement that we are modifying, the + sign the new statements. In this case, the new statement represents the declaration of a new variable. The second new statement represents the modification of an existing variable.

Note that in the previous example we used the operations introduced in Section 1.2. We define a language with primitive operations {*read, write, search, compare*}, which are operations often used in databases. By updating the state of the data structures in the system, we move a step forward into achieving hot-patching. We are advancing the field of hot-patching by focusing on one of the main obstacles for dynamically updating applications, which is resolving conflicts obtained by updating data structure semantics. Some examples of the patches that we are able to hot–patch their data semantics are shown in Figure 1.2 and Figure 1.3. Also, in Figure 1.4, we illustrate the process of hot–patching data structures using DPL.

One of the main advantages of using a database approach, when defining our set of operators for constructing a **data patch**, is having basic and primitive operations in our language that follows principles of relational–algebraic databases (such as MySQL). This is a

```
                smb_panic("make_connection: PANIC ERROR. Called as nonroot\n");
        }

+       if (conn_num_open() > 2047) {
+               *status = NT_STATUS_INSUFF_SERVER_RESOURCES;
+               return NULL;
+       }
+
        if(lp_security() != SEC_SHARE) {
                vuser = get_valid_user_struct(vuid);
                if (!vuser) {
```

Figure 1.3: Example of a source code patch. Similarly to Figure 1.2, the + statements are new statements for the application. In this case we are modifying the value of a variable by following a condition (i.e., if the condition meets, then modify the variable). We also change the control flow of the program when returning NULL.

basic concept for language theoretic security [2], where using a well-defined language assures its security.

1.3 Thesis Scope

Having to restart an application after updating is not a new *problem*, but it still remains a very important, and difficult one. When we hot–patch an application, we are not only introducing new source–code but also adding or changing control flow and data semantics; this can introduce conflicts into the new state of the application. We think the solution we present in this thesis is novel, and is directed to solve the conflicts introduced when changing the data semantics of an application after dynamically updating it with a security patch.

We must also clearly state what is the range of operations that our system is capable of updating. The purpose of the system proposed in this thesis is to execute our design and demonstrate how it is capable of updating data structures from a running application. The techniques that this thesis presents can be used to help advance the field of hot–patching, or dynamic software updating, by introducing a new technique focused on updating the new data semantics introduced by a patch.

Our system is not yet capable of updating complex applications (e.g., the ones which security patches we are studying) because of technical limitations. However, the purpose of

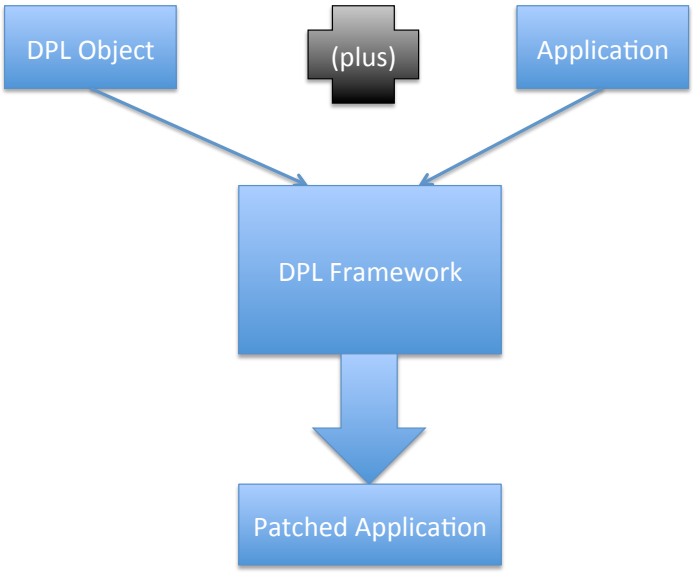


Figure 1.4: An illustration of how DPL patches an application by getting as an input the DPL object and the application to be patched. By the end, the application's data structures are updated according to the data patch *without* interrupting the process.

our proof of concept is to demonstrate the three main steps in this thesis: importing data structures from an application, patching or modifying the data structures, and exporting them back to the application to update its state. We demonstrate this by using different experiments we explain in Chapter 7, and use these steps to hot–patch data structures of a small test suite we present in Chapter 8.

The approach that we take on this investigation integrates elements from systems security, hot–patching, database systems, machine learning theory, dimension–reduction, and assembly-level languages.

1.4 Motivation

The issue of restarting applications after being patched has been studied by different authors over the last decade [38, 35, 57, 49, 60, 46, 56, 13]. There are a lot of studies on dynamically updating applications (i.e., without restarting them) under different names (e.g., Hot Patching [49], Dynamic Software Update [19, 60], Self-Healing [44]). However, at least to our knowledge, there is no framework that is capable of correctly patching an arbitrary application, therefore restarting applications after patching them remains an issue.

We assert that some developers of these frameworks do not take into consideration many conflicts that arise from hot–patching an application. In this thesis we are not studying how to dynamically modify or update source code, we are focusing on how the data semantics of an application are modified once applying a patch. This is the main motivation of this thesis.

In other words, we study the conflicts that security patches may introduce, with particular attention to how such patches interact with the existing state of data in an application.

1.5 Thesis Statement

The main purpose of this thesis is to advance research in the hot-patching field further by introducing a new technique to study the data semantics that a patch introduces to an application. Dynamically patching existing security–related flaws presents two related problems: (1) triaging the data semantics of the code patch, and (2) patching the line code and data in the application.

Thesis statement: *It is feasible to construct a model of common data–patching operations and apply combinations of those operations to real applications in order to update statements that are modifying data structures to fix flaws related to software vulnerabilities.*

We study how statements that modify data structures introduce new data semantics to the application. Based on these studies, we design and implement a framework that successfully patches the data semantics of an application when dynamically applying a **data–patch**.

1.5.1 Hypotheses

We are focusing on dynamically updating what is considered to be one of the main conflicts occurred when hot-patching: the data semantics being out of date. To do so, we need to understand how a patch modifies data structures and be able to update them with our system. The set of hypotheses we will be studying in this thesis is:

Hypothesis A: *Security patches, typically, modify a small set of statements related to data structures.*

Hypothesis B: *We can classify statements that are modifying data structures into different categories that we define as data operations.*

Hypothesis C: *We can use these data operations to dynamically update existing data structures according to how security patches modify data semantics.*

Hypothesis D: *DPL has enough computational power to update the data semantics of a set of programs written in the C programming language.*

1.6 General Research Questions

Based on the previous hypotheses, we decided to answer the following research questions:

1. By studying our set of 75 security patches, can we generalize statements modifying data structures by selecting a set of common data operations?
2. How many security patches are we able to express using our set of data operations?
3. Are we able to automate the task of selecting patches that are feasible to translate to our set of data operations?
4. Can we design, and implement a system for updating applications according to our set of data operations?
5. What is the performance and overall evaluation of the aforementioned system?

This, however, is not our final set of research questions. In further chapters we expand this set of research questions by adding specific research questions that we think our experiments address.

1.7 Contributions

To the best of our knowledge, there are no measurements of how data is modified by a patch. Our work provides evidence to support that we can classify patches according to feasibility and difficulty of applying them to an application by taking into consideration data modifications. We also study if we are able to apply machine learning techniques

to our dataset and present results according to our evaluation. Besides, we present an implementation of a proof of concept for DPL, a framework capable of hot–patching data structures of an application according to how a security patch modifies them. This proof of concept is able to patch basic running applications using the heuristics explained in this thesis. We explain the design of each application in Table 8.6 of Chapter 8.

1. We provide a set of data operations that describe common statements, represented in security patches, to modify data structures. We call these data operations Data Modification Machines (DMMs) and they work as a classification for different types of data modification statements we encountered in our study of security patches.

2. We manually classify 75 security patches of different open–source projects (including Apache HTTPD, Samba, and Asterisk) into feasible or infeasible patches depending on whether or not they are translatable to our language of data operations. We found 38 of these patches to be feasible and 13 infeasible to implement using our set of DMMs. The remaining 24 patches do not modify data structures when applying them to an application.

3. We automate the task of classifying security patches by using machine learning algorithms (Support Vector Machines) to predict if a security patch is feasible to translate into our set of data operations.

4. We propose a proof of concept for a system capable of dynamically updating data structures of different C applications according to a security patch.

5. We present an analysis, and implement our set of data operations using the DPL system. We also provide a description of how DPL patches the data structures of an application according to 5 different security patches. Our sys-

tem successfully uses our DMMs to hot–patch the data structures of simple C applications, according to security patches, with a low performance overhead.

6. As a corollary, we propose different applications for DPL including its functionalities as a hot–patching framework, and a data structure modifier. In the future, we plan to deploy DPL as a system for patch developers to test if a patch could introduce conflicts when being hot–patched.

1.8 Thesis Overview

The remaining of this thesis is presented as follows:

In Chapter 2, we present the related work of the thesis and provide background material of what our work is addressing. This includes related work in hot patching, data structure modifications or recovery, and machine learning techniques applied over source code.

In Chapter 3, we provide a design of our approach for solving our main research question. This section is divided into three different subsections explaining the design of the three main components of this thesis. These components are: manual analysis of patches, applying machine learning on patches, and DPL proof of concept.

In Chapter 4, we present the manual analysis of security patches. This manual analysis consists of translating patches to a primitive language, then to a graphical representation of this language, and finally to a feature vector that works as an input for our machine learning algorithm. We also manually classify the patches as feasible or infeasible and attach a respective classification label (i.e., feasible or infeasible) to each feature vector.

In Chapter 5, we present a machine learning approach for classifying patches as feasible or infeasible. By using support vector machines, we provide them the feature vectors defined in Chapter 4 as an input and get the feasibility of the patch as an output. Finally, we present the results of the SVM prediction for this classification.

In Chapter 6, we explain the implementation of a framework for DPL capable of per-

forming the import and export routines, the Data Structure Recovery (DSR), and the live modification of data structures according to the data patch object.

In Chapter 7, we present an evaluation to demonstrate the capabilities of the DPL System. The test suite demonstrates different data modification statements that were studied when analyzing the set of patches.

In Chapter 8, we discuss the results of the manual analysis and translation of patches, machine learning techniques applied to patches, and the evaluation of the DPL system.

At the end, we conclude this thesis, and present suggestions for future work.

Chapter 2

Background and Related Work

The technique of patching applications without stopping a process has been extensively studied in the past years. Even though there has been progress in the last decade, it has not been deployed for commercial applications or systems yet. Several authors [] have addressed this issue by developing frameworks for dynamically patching an application in many different programming languages. Other authors focus on hot–patching kernels, a more specific approach than hot–patching applications since some authors implement kernel modules to do the task automatically [4, 5]. Patching applications without restarting the process after it has been updated is a technique studied by authors under different names. Common names for this technique are dynamic software updating (DSU), self–healing, and hot–patching.

Besides these techniques as related work, DPL is also related to other concepts of computer science such as machine learning algorithms, databases, and Binary Instrumentation. This chapter starts by giving an overview of the background concepts we think are necessary for understanding subsequent chapters. We then analyze different research that is related to the objectives of this thesis and explain at the end how our heuristics differ from others.

2.1 Security Patches

Applying security patches is a common defensive practice for modern software systems. They offer a quick fix for vulnerabilities encountered in already deployed software applications. This eases the process of updating an entire application by isolating the piece of source code that has the vulnerability. Our framework consists of a manual workflow where we construct a data patch object according to the source code we analyze, and an automatic workflow where we import, update, and export the data structures of different programs.

For this thesis, we consider the isolation of operations that security patches provide as one of the main advantages for hot–patching data structures. As an example of an isolation of operations provided by a patch, we can refer to Figure 2.1 which shows a security patch that fixes a vulnerability in the open–source project Samba. This security patch focuses on fixing a *particular* if–case from the application that brings a vulnerability that allows local users to modify the membership of Unix groups. Many security patches studied in Chapter 4 take this approach – patching a constrained operation – when fixing vulnerabilities. Our hypothesis A in Section 1.5.1 looks for a mechanism to translate these operations into a set of data operations, taking advantage of the constrained scope security patches modify on software systems.

2.1.1 Common Vulnerabilities and Exposures

Releasing software is a complex, hard, and – many times – a hasty activity; human beings are not perfect, therefore developers make mistakes when releasing their software. These mistakes could lead to vulnerabilities that attackers target. Some examples of these vulnerabilities are zero-day vulnerabilities [8] that many security engineers have found in early releases of applications, operating systems, and service packs. Many open source projects obtain feedback from security engineers or other developers about possible vulnerabilities that were found in their source code. This feedback helps for creating a **security patch** to fix these vulnerabilities. We define a security patch as **a piece of source code designed to fix a security issue, by modifying the application's functionality, or the application's semantics**. Figure 2.1, illustrates a few concepts from the patch:

- As its name says (CVE-2008-3789), he patch was deployed in 2008.
- It addresses the vulnerability CVE-2008-3789[1] explained in Figure 2.2.

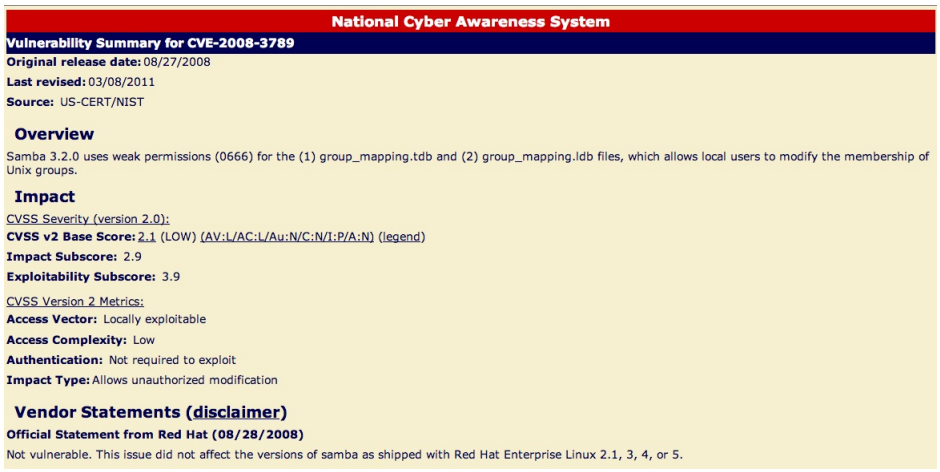
[1]CVE stands for: Common Vulnerabilities and Exposures, and it is a database of publicly known information security vulnerabilities and exposures.[64]

```
From 2eaf4ed62220246bcc1a9702166b0b4f381fdae3 Mon Sep 17 00:00:00 2001
From: Andrew Tridgell <tridge@samba.org>
Date: Wed, 27 Aug 2008 10:45:43 +0200
Subject: [PATCH] ldb: Fix permissions of group_mapping.ldb.

This one fixes bug #5715 and CVE-2008-3789.
(cherry picked from commit a94f44c49f668fcf12f4566777a668043326bf97)
---
 source/groupdb/mapping_ldb.c |    8 +++++++-
 1 files changed, 7 insertions(+), 1 deletions(-)

diff --git a/source/groupdb/mapping_ldb.c b/source/groupdb/mapping_ldb.c
index 6775f61..ce65d7c 100644
--- a/source/groupdb/mapping_ldb.c
+++ b/source/groupdb/mapping_ldb.c
@@ -74,7 +74,13 @@ static bool init_group_mapping(void)
        if (ret != LDB_SUCCESS) {
                goto failed;
        }
-
+
+       /* force the permissions on the ldb to 0600 - this will fix
+          existing databases as well as new ones */
+       if (chmod(db_path, 0600) != 0) {
+               goto failed;
+       }
+
        if (!existed) {
                /* initialise the ldb with an index */
                struct ldb_ldif *ldif;
--
1.5.4.4
```

Figure 2.1: An example of a security patch developed for Samba to fix vulnerability CVE-2008-3789. As the Figure illustrates, it focuses on fixing a particular if case. This is what we called an *isolation* offered by security patches.



National Cyber Awareness System

Vulnerability Summary for CVE-2008-3789

Original release date: 08/27/2008

Last revised: 03/08/2011

Source: US-CERT/NIST

Overview

Samba 3.2.0 uses weak permissions (0666) for the (1) group_mapping.tdb and (2) group_mapping.ldb files, which allows local users to modify the membership of Unix groups.

Impact

CVSS Severity (version 2.0):

CVSS v2 Base Score: 2.1 (LOW) (AV:L/AC:L/Au:N/C:N/I:P/A:N) (legend)

Impact Subscore: 2.9

Exploitability Subscore: 3.9

CVSS Version 2 Metrics:

Access Vector: Locally exploitable

Access Complexity: Low

Authentication: Not required to exploit

Impact Type: Allows unauthorized modification

Vendor Statements (disclaimer)

Official Statement from Red Hat (08/28/2008)

Not vulnerable. This issue did not affect the versions of samba as shipped with Red Hat Enterprise Linux 2.1, 3, 4, or 5.

Figure 2.2: Example of a CVE entry. This CVE explains the vulnerability that the patch in Figure 2.1 is fixing.

- It is modifying 7 lines of source code where 2 lines are empty spaces, 2 lines are comments and **3 lines are statements**.

- Of the statements being modified, it is possible that the variable *db_path* gets modified by the chmod function. Our system doesn't handle this type of operation because it depends on how the external function could potentially modify the variable. However, we considered these statements when analyzing data modification statements in Chapter 4.

Other terms that are related to security patches are **vulnerability** and **exposure**. Quoting from the Common Vulnerabilities and Exposures (CVE) [64] website:

> An information security "vulnerability" is a mistake in software that can be directly used by a hacker to gain access to a system or network.
>
> CVE considers a mistake a vulnerability if it allows an attacker to use it to violate a reasonable security policy for that system (this excludes entirely "open" security policies in which all users are trusted, or where there is no consideration of risk to the system).
>
> An information security "exposure" is a system configuration issue or a mistake in software that allows access to information or capabilities that can be used by a hacker as a stepping-stone into a system or network.
>
> CVE considers a configuration issue or a mistake an exposure if it does not directly allow compromise but could be an important component of a successful attack, and is a violation of a reasonable security policy

2.1.2 Limitations of Security Patches

Many times, when deploying a security patch, developers aim to fix conflicts related to the semantics of the application – making security patches hard to deploy without restarting the system to a known state.

In this thesis, we focus on how the new semantics a patch brings to an application modify components such as data structures declared in the application. This makes the application to be **out of date** with the new state of the application after applying a patch. With this, we can refer back to our thesis statement in Chapter 1 and focus on updating the data structures to be *in harmony* with the new semantics entailed by a patch.

2.2 Components of DPL as a System

This thesis explains the design, implementation, and evaluation of the Data Patching Language (DPL) System. DPL is a system that **hot–patches data structures** according to a novel construct we call a *data patch*. DPL achieves hot–patching by attaching to a process using the Pin dynamic binary instrumentation library [37]. After attaching to the process, DPL parses the debugging information entries (DWARF [66]) of the application and imports every data structure into a database. For storing the data structures into the database, DPL uses a MySQL API offered for the C language. Then, DPL exports the data structures back to the application and permits the process to continue running, therefore the process does not need to be restarted. DPL achieves a dynamic update of the data structures of the application while minimizing downtime or service disruption.

In this thesis, we assume that the information of the data structures (e.g., size, type, name, value) is available either in the ELF [62] or a net service that the developers provide for recovering the data structures of their application.

2.2.1 DWARF information

DWARF is an Application Binary Interface (ABI) standard that offers a detailed description of the debugging information of a particular application. With DWARF information, DPL obtains the address, data, name, and type of every data structure declared in the application. An advantage of using DWARF is that it labels the data structures according to the scope where they were declared. Therefore, we are also able to organize data structures according to the functions where they were declared. This organization is convenient for dividing data structures according to their scope in the application inside our database.

We worked with the Version 4 of the DWARF information. The API we used for parsing the DWARF information is *libdwarf* [66].

2.2.2 Pin: Dynamic binary instrumentation

Another component of DPL is an interface that communicates with the Pin [37] library. Pin is a dynamic binary instrumentation framework that offers a library and runtime compiler to allow the creation of different tools to dynamically manipulate an application. This manipulation includes the analysis of the architecture, and the injection of arbitrary source code into the application. The C source code we inject into the application is related to the heuristics used to hot–patch data structures that we explain in Chapter 6.

2.3 Related Work

DPL as a framework is related to several topics from computer science including shape–analysis, hot–patching, self–healing, and binary instrumentation. As stated in the previous chapter, DPL consists of three steps: the import routine which stores every data structure of the application into a database, the modification of data structures using a data patch object, and the export routine that updates the data structures. These steps are further explained throughout the thesis, more specifically in Chapter 6.

This section highlights how DPL relates to many other sub–fields of computer science and how our approach differs from previous approaches aiming to successfully implement a framework for hot–patching applications.

2.3.1 Self–healing mechanisms and Langsec

The concept of hot–patching can be compared with software self–healing because we are dynamically updating (i.e., healing) a process, or application. Software self-healing has been an active area of research in the last decade. A fundamental paper that studied the philosophical aspects of software self–healing and its implications, was presented by Locasto [36] in 2007. This paper denotes one of the fundamental problems of security in computer science, which is that program designs lack a complete description of how to handle errors. Which brings us to the concept of *LANGSEC* [2], that is also related to this thesis because **by analyzing a description of an application** (the DWARF info [66]) **we were able to handle errors** (i.e., update applications according to the description). Another principle defined in LANGSEC is the analysis of arbitrary computation in order to handle errors. Arbitrary computation is one of the key problems when trying to update data structures according to a security patch. It brings computation that we, or a machine, are not able to predict correctly. We think that by applying the concepts of LANGSEC we are able to improve our system, and other hot–patching systems by maintaining a discipline that tells us about situations we cannot deal with.

Other approaches analyze the hardware architecture [54], or emulate sensors that detect anomalies in source code [48, 52, 55, 65, 43], or hot–patch OS kernels [5, 11]. However, none of these methods has been completely achieved or implemented in modern systems. We are not aiming to develop a final product capable of being deployed as a framework for hot–patching. The approach we are taking in this thesis is to attack one of the main obstacles for achieving hot–patching, which is data structures being out of date with the new semantics after patching an application.

Even though our system has limitations, we are planning to deploy it as a test system for patch developers in the future. This way, patch developers can study how a patch modifies data semantics and tune the patch to be feasible to implement in a hot–patching framework. This follows the principles of langsec because our system will evaluate computation that has not been tested on large systems to see how semantics get modified and handle it to obtain a secure and safe environment (i.e., the patch being feasible and safe to hot–patch).

2.3.2 Hot Patching and DSU

The technique of patching or updating an application without interrupting it or having to restart it afterwards is known by many different names including hot patching and dynamic software updating (DSU). There is a plethora of work [29, 60, 46, 49, 4] on this technique and many authors are trying to address it by proposing different frameworks or tools. These frameworks approach hot–patching the source code of an application according to a security patch. Many authors of these frameworks have stated that dealing with complex statements such as shared–library updates, data structure modifications, type definitions, and changes to function prototypes makes hot–patching difficult to achieve [4, 49]. After analyzing our set of 75 security patches, we concluded that many of these patches modify the aforementioned types of statements. In this thesis we focus on one of these types: statements that are modifying data structures.

One common mechanism to address hot–patching is by proposing frameworks such as [4, 6, 12, 39] to target OS Kernels. Arnold et al. [4] presented Ksplice, a system capable of hot–patching the kernel according to security patches and, in some cases, by introducing new lines of code to the security patch. They studied conflicts that security patches present when modifying data structures and concluded that patch developers can correct most security vulnerabilities without making any semantic changes to persistent data structures. They also found it inevitable to add new lines of code when data structures are modified by a patch. The purpose of the new lines of code is to specify the semantics of these statements,

for example in some cases modifying a field of a data structure implies modifying many instances of that data structure, therefore Arnold et al. include these semantics as new lines of code for the patch. However, it is hard for patch–developers to fix every vulnerability without eventually changing a data structure. This is why we study the problem of hot–patching data structures on its own.

Da Silva et al. [5] uses a different approach by implementing a dynamic update framework into an already existing operating system (K42). They were able to dynamically update large data structures by using a technique called lazy update [29], where data structures were patched at runtime as needed. However, the scalability of their framework is different than ours, since theirs was directly implemented according to a specific OS. We used our framework to hot–patch more general small applications that have in–memory resident data structures inside. Therefore, we decided to update all data structures at once instead of using a lazy update.

Other authors, such as Candea et al. [10] address this issue by issuing micro-reboots which is another mechanism for hot-patching, it updates components of the application without interrupting the rest of the application. On the other hand, many different authors [25, 42, 3, 26] address hot patching by targeting applications. These scientific papers approach hot–patching by implementing a framework capable of dynamically updating source code. However, none of these frameworks are able to correctly patch the new data semantics introduced by the patch. Many authors have issues trying to update local variable modifications or more complex data modification statements. We address many types of data modification statements in this thesis, including local variable modifications.

Even though there is a plethora of work on the field of hot patching, it still remains to become a *mainstream* technique. Many operating systems still need to be restarted after an update due to inconsistencies in their states. This is why many authors [42, 27, 24, 9] address hot patching in an *update-point* approach (i.e., by updating the application only

at certain points of its execution) and only updating **when the code and data to be patched are not being executed**. DPL can be executed at any point while running the application, but it still has some technical limitations that makes the task of patching an arbitrary application difficult to achieve. Nonetheless, we have to remember that DPL was designed and implemented to **aid hot patching** by studying how to correctly patch data structures without getting inconsistent states.

2.3.3 Genetic Programming

There is also some work on genetic programming [65, 22] on where a program understands how to dynamically patch itself by evolving its source code to fix the error. However, this approach is very different from previous approaches because of the lack of a security patch. Applying genetic programming for hot–patching applications assumes that the application is capable of hot–patching itself. But the case studies for these approaches are very limited for successfully achieving hot–patching in an arbitrary application.

We think that collaboration between patch and application developers is the key for successfully deploying security patches that are hot–patch*able*.

2.3.4 Data Structure Modifications and Shape Analysis

The main approach of this thesis is to update data structures according to security patches. The C programming language offers a rich and complex environment with many ways to declare data structures. There are many techniques (e.g., different techniques for creating a linked list or a tree) and many scopes (e.g., global variables, dynamically allocated variables) to declare variables. DPL's task is to correctly deduce and parse every data structure declared in the application. There are many approaches to deduct these data structures and these mechanisms are going to be explained in this subsection. To see the heuristics used for this thesis we can refer to Chapter 2 and 6.

Cozzie et al. [17] were capable of, by having a memory image of an application, detect and

deduce its data structures. One of the main problems that they found, when implementing their system Laika, is that they could not define an ending point for pointers, therefore they estimated an ending point for the structures pointed. We were able to find *ending points* for pointers by having the properties of the data structure (given by DWARF) they pointed. The biggest similarity between Cozzie's work and mine was the use of a memory image for deducing data structures. We used Pin in a similar fashion to store the data structures into my database. Other shape analysis work can be found in Corbett's research [16] and Manevich et al. [40], which aims to study a higher-level approach than Cozzie's by studying the set of allocation instructions that modify the heap of the application – although Corbett's work was more focused on object-oriented concepts offered by Java.

Shape analysis is a broadly studied concept, some of the most prominent work dates back in 1996 when Ghiya et al. [23] and Steensgaard [58] started studying how pointers that point to the heap can be used for deducing the type of data structure they are pointing to by using recursion.

The concept of analyzing structures using pointers, to deduce the type of data structure and its values, is used in the field of shape analysis. For DPL, we analyze pointers by using the DWARF information [20] and dereferencing the addresses of the members of the data structures. By the end of this thesis, we are able to deduce linked lists and other data structures defined in our test cases and we are able to store these data structures into different tables in the database. Other work in data structure analysis is by Chilimbi et. al. [15] where they study, from the point of view of programmers, how to improve the locality of pointer–based data structures. This is a challenge we also encounter in our research, because there are many ways to define data structures in C, and thus many ways to parse the debugging information of data structures.

2.3.5 Machine Learning Techniques on Source Code

Applying machine learning techniques to the problem of patching is, to the best of my knowledge, something new to the machine learning field. There are many machine learning techniques that have been applied in the security field before (especially in the intrusion detection field), but we could not find any implementations of machine learning over patches. We decided to use Support Vector Machines (SVMs) using R [31] as our algorithm because it is a common technique used by other approaches for learning from source–code [50, 59, 63].

However, classifying source code with a signature-based approach is similar to how our SVM classifies a patch. Having an intermediate language is used for applying machine learning techniques on virus analysis [50, 59] and similarly with spam analysis [63], where spam is classified according to a pre-determined set of keywords and virus analysis with a signature based approach. The similarities between spam and viruses with our work is that they also use intermediate language to express source code. This is the main reason why we decided to use SVM as our machine learning algorithm for classifying our dataset. By having an intermediate representation of our patches, we get 15 different features explained in Chapter 4. We then represent these features as a 15-Dimensional vector that works as input to our SVM.

2.4 Unaddressed Challenges

A limitation we find in current DSU/hot–patching systems [60, 46, 49] is that they go *straight to the point* by designing and implementing some heuristics for hot–patching source–code. We found this to be a very difficult principle to follow when trying to achieve hot–patching. We think of hot–patching as an art, or technique that requires some previous analysis – this is why we decided to first **study** our set of patches and **then** create our framework. The detailed analysis of our patches explained in Chapter 4 helped us to address some challenges that we think most hot–patching frameworks do not meet. Some of these challenges are:

Implement General Data Operations for Translating Source–Code Statements

Our empirical study in Chapter 4 helped us understand the semantics of operations related to data modifications introduced by a patch. We classified these operations into several groups of statements according to the operations they are addressing to patch.

As an example, many security patches modify if cases by introducing new variables, changing their if–conditions, or modifying existing variables inside the scope of the if–case. We classified these operations as a particular data operation that modifies the heuristics of if–cases. These data operations were called Data Modification Machines (DMMs) and are explained in Chapter 4. We finally implemented our set of DMMs in Chapter 7.

Automatically Classify Security Patches

According to the results of several authors [25, 42, 3, 4], some patches are going to be harder to hot–patch than others. Moreover, some patches are going to be infeasible to hot–patch because of the complexity of the operations being updated. We decided to automate the classification of patches as feasible or infeasible to hot–patch according to the type of statements they are modifying. These classification helped us understand what type of data modification statements create more conflicts when hot–patching an application, and constrained us to a set of feasible patches to experiment with our DPL System.

Hot–patching Data Structures

The main challenge that our DPL System addresses is the task of dynamically updating the data structures of an application according to a security patch. This updates the application to a new state according to the data semantics introduced by the patch. We implemented a framework named Data Patching Language System (DPL System) that is capable of instrumenting over an application to dynamically update its data structures. To the best of our knowledge, there are no other implementations of hot–patching data semantics of an application making it our main contribution in this thesis.

Chapter 3

Design Overview

In this chapter, we explore the design of the workflow in this thesis: our empirical study of security patches, automating their classification as *feasible* or *infeasible*, and implementing our proof of concept for the DPL system. Our empirical study consists of a manual analysis of our set of 75 security patches. With it, we study the operations involved in applying a security patch to an application. We analyze the set of patches in order to understand how data structures were modified by a security patch, and how to apply those changes using the DPL system. After studying the patches, we created our concept of data operations (which we called DMMs in Chapter 4), we also started manually classifying patches as *feasible* if we are able to translate them to our language, or *infeasible* otherwise. At this point, we started questioning if it was possible to automate this classification of security patches by using machine learning techniques. We decided to use Support Vector Machines (SVMs) for doing the task of classifying patches according to what the machine learns from our manual classification. Finally, we used a subset of the patches predicted as feasible for hot–patching data structures on simple application. These experiments are presented in Chapter 7.

3.1 Workflow and Experiments

The workflow of our analysis of patches consists of seven main stages. We provide a brief enumeration of this workflow here and follow it with a high–level explanation of the major steps. This workflow consists of a procedure that begins with the analysis of patches, continues with the application of SVM and PCA [30], and ends up with getting results by applying the testing dataset to our SVM. Finally, we discuss the workflow involved in the design of our DPL system.

We first gathered 100 different patches from open source projects such as Apache, HTTPD, Asterisk, and Samba. Most 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. The analysis of the security patches does not only consist of counting the number of lines introduced by the patch, but also consists of a detailed analysis of the type of statements the patch is introducing. This analysis includes recognizing the statements that are modifying data structures and translating them to our language. If we are capable of translating the patch into our four data–operations then we are able to update its data semantics. Otherwise, we are not able to update them using our language.

Then, we manually translated each patch of the training and testing datasets into their graph representation, according to the heuristics defined in Chapter 4. We then translate each graph into a vector that represents all of the attributes, or features, from each graph. This is considered to be our initial feature vector. We also present the definition of Data Modification Machines (DMMs) as a group of operations resulted of classifying data operations according to how they modify data structures.

We apply PCA over the initial feature vectors to see their correlation. PCA gives us the most independent features which are then considered to be our final feature vector. We manually traced 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.

By having a final feature vector of all the patches in the training and testing datasets, we gave 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. We then apply our testing dataset, without any labels, as a feature vector matrix. Our SVM gives a prediction on feasibility according to what it learned. Three different experiments were studied over this set of patches.

Experiment 1: We select 50 patches that represent our learning data and 25 that represent our testing data. We decided to use these numbers so the majority of the patches is in the learning dataset. We then did the opposite in our second experiment and compared both results. We manually categorize each patch as feasible or infeasible, and apply the learning dataset to a Support Vector Machine (SVM) algorithm. We then provide the testing dataset to the SVM and get the prediction results according to what the SVM learned about the dataset.

Experiment 2: Similarly to Experiment 1, we also provided a learning and testing dataset to the SVM. This time we use different combinations of the dataset by randomly selecting 30 data points as the learning dataset and 45 data points as the training dataset in five iterations. The majority of the patches is this time in the testing dataset.

Experiment 3: This experiment consisted of a leave–one–out cross–validation approach. Our intent with this experiment is to evaluate each patch as a testing dataset and use the rest of the patches as learning dataset. The results for this experiment were consistent with Experiments 1 and 2.

After getting the results from our SVM, we gathered the patches that our SVM predicted correctly (i.e., was predicted to be feasible by both our analysis of patches and the SVM) and use them for our experimental methodology – the experimental methodology is explained in Chapter 7.

The implementation of our proof of concept for the DPL System consists of four main stages. These stages or steps define the process of hot–patching data structures using our system:

The first step is the creation of a Pin tool that is capable to attach to a running process and inject arbitrary C/C++ code. Inside the Pin tool, we designed a communication with a MySQL database by using the MySQL C API. The second step of the design of the framework is the importation of every variable and data structure from the running process

to the database. This was divided into:

- Global Variables: Every globally declared variable and data structure was included in a table called "Global_Variables".
- Local Variables: Every variable that is currently on the stack was included in a table called "Local_Variables".
- Dynamically Allocated Variables: Every variable that was dynamically allocated (by using a memory allocation function from C), was included in a table called "Dynamic_Variables".
- Data Structures: Every instance of a data structure, such as a linked list, detected by our framework was included in a table named as the structure. Examples for how we stored these data structures in our database are presented in Chapter 6.

Then, we update the database using a set of SQL statements in the form of a data patch object to be consistent with the semantics introduced by the patch. Finally, we export all the data structures and variables back to their respective addresses according to the database metadata. After updating the variables inside the program, the Pin tool detaches from the application and the process was updated according to the data patch without being restarted or interrupted. We then experiment with DPL by modelling the DMMs and some of the security patches predicted as feasible by the SVM.

3.2 Manual Analysis of Patches

The main goal of this thesis is to characterize what makes a patch feasible, according to our definition of feasibility in Section 4.3, and use the feasible patches to evaluate our proof of concept. The mechanism we use to characterize these patches is by analyzing a dataset

of patches and stating how many patches out of the dataset modify the heuristics of data structures. This would tell us a percentage of patches that could result in conflicts when using a hot–patching framework if the data structures of the application are not updated after applying the patch.

We can see this analysis as an empirical study that helps to understand how patches modify data structures, which statements or operators are commonly used to modify them, and to obtain a set of statements that represent algorithms that we can use to update data structures. We define this set of algorithms as the set of Data Modification Machines (DMMs), and they are explained in Table 4.1.

With this analysis of patches, a first research question arises: How many of these patches are we able to express in the form of a data patch?

We could also see a data patch as a group of statements that uses primitive algebraic operations such as **read, write, compare,** and **search**[1] to correctly patch the data with the new semantics that the patch implies. We can define a patch to be feasible to implement as a data patch when we are able to express the data modification operations using the previously mentioned set of operations. We also want to study if there are any other variables that define feasibility on a patch, such as its control flow, lines of code modified, or other variables that we suspect define feasibility on a patch. We aim to automate the process of defining feasibility on a patch, therefore we conclude that one automation mechanism can be applying machine learning algorithms to the dataset.

3.2.1 Translation of Source Code into Graph Language

Several authors[50, 59, 63] have addressed the application of machine learning techniques over source code by using an intermediate language. A common technique is to translate source code to a graph language, such as an abstract syntax tree (AST), and then apply a

[1]We can think of these operations as the ones we are capable to perform over data in a database, which is going to be used by DPL.

machine learning algorithm to the graph. However, ASTs did not fit well enough with our purposes. We wanted the nodes of the trees to represent different types of statements and to be able to generalize statements according to their purpose. Each type of node is further explained in Table 4.1 of Chapter 4.

We decide to use SVMs as a machine learning algorithm because it is the most commonly used algorithm over source code and intermediate graph language. However, SVMs need a feature vector as an input. For getting this vector we followed several steps by answering the following questions:

1. How many and which features can we get from a graph? ASTs have many features that can be considered for our purposes. These features are further explained in Chapter 4.

2. Can we hot–patch data structures according to the control flow of the graph? Because each node represents the heuristics of how data needs to be patched, and the leaves of the trees represent the data modifications, we can think of the control flow of the graph as the control flow of data operations in a patch.

3. From the set of features, which features are the most independent and tell us more about the graph? For answering this question, we used a dimension-reduction technique called Principal Component Analysis (PCA).

3.2.2 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 features of a patch according to its data modifications, and our four data operations. These features include the number of write, search, read and compare operations, function calls returning variables, and the number of operations inside a loop.

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.

3.2.3 Feasible and Infeasible

We categorize patches as "feasible" or "infeasible" depending on the ability to update the data structures of the application to be consistent with the new semantics introduced by the 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 by our definition of feasibility.

Feasibility in our work is defined as **being able to correctly update every data structure in an application according to the new semantics implied by the patch**. However, if we are not able to update the data structures according to the patch, we think that hot patching it would be infeasible because of inconsistent states of data structures.

For our studies, we defined feasibility to be a binary decision of *yes* it is feasible or *no* it is not, this decision is resulted after analyzing a 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.

For a statement to be feasible we need to be able to translate it into simple data manipulation operations. We defined 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.

Most of the patches we found to be "infeasible" were because of expected user input,

radical 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). The concept of feasibility is further explained in Section 4.3.

3.2.4 Labeling

We must first label each member of our training dataset as feasible or infeasible. We manually study each patch in our training dataset and analyze its feasibility. This is done by trying to model each patch's data modification logic using relational algebraic operations.

This labeling activity is a prime example of the difficult and time–consuming nature of patch analysis and classification. This limited our research 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 cannot be automated. 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 known algorithm to determine this. The process of preparing our data to work as input for the SVM is explained in Figure 3.1.

3.3 Applying Machine Learning on Patches

The process of classifying a patch as *interesting*[2] is a manual process that requires careful study of the patch that we want to apply. Moreover, not every patch with statements that modify data structures can be patched. This brings us to classify patches as feasible or infeasible, according to the feasibility of updating data structures in the application to be

[2]A patch is interesting to our framework when there is a need to update the data structures of the application after applying the patch.

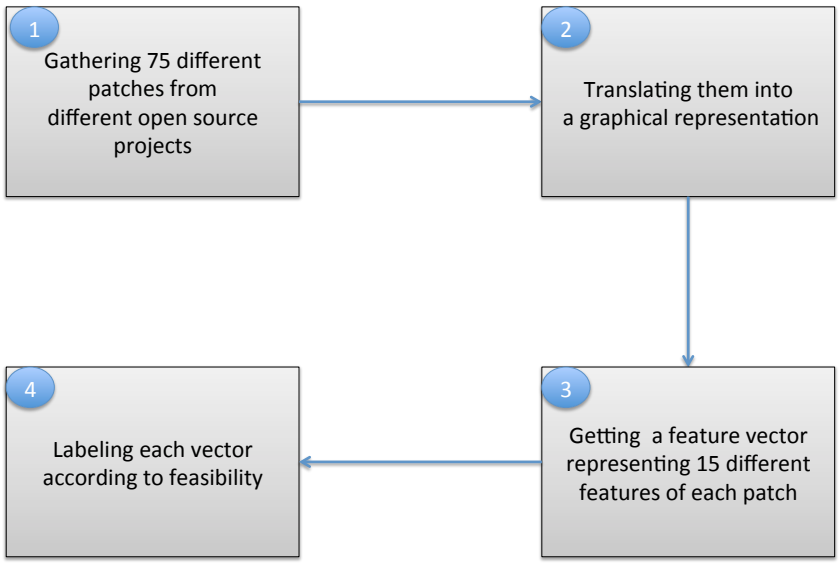


Figure 3.1: Diagram for the process of analyzing and classifying patches.

synchronized with the patch. By using machine learning techniques, we could automate the process of classifying a patch as feasible or infeasible. For future work, we plan to include more researchers to study these patches as a cross–validation technique to compare their results with our results.

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 [51] 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.

After translating each patch into its vector form and labelling it, we use it to train our SVM. We obtain a hyperplane and study a possible relationship between features and feasibility. The rest of the patches work as a testing dataset and they were used to see how

well the hyperplane classifies a patch.

3.3.1 Applying PCA over data

After translating the patches into their ultimate form (a feature vector), we get a 15-dimensional vector which brings a lot of noise and could bring false positives and false negatives to our results, the 15 features are explained in Chapter 4. Also, a 15-dimensional feature vector is hard to map into a vector space, that is why we decided to apply a dimension reduction technique called principal component analysis (PCA).

PCA is a statistics technique that reduce the size of a set of features according to their co-dependence. More precisely, by having our set of 15 different features, we can map them to a set of lower features that highlights the most independent set of features. When applying PCA, we mapped our 15-dimensional vector to a 4-dimensional. This not only reduces the noise and bad results of our dataset, but also makes easier and faster the process of obtaining the four features for having a feature vector of a patch.

3.3.2 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.

The focus of our study is whether there exists a relationship between the set of features for each graph and our definition of infeasibility 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."

After studying seventy-five different patches, we came to the conclusion that, in order to correctly label these patches, a careful and long manual study needs to be made. This process could be shortened if we find some mechanism to express a patch as an input for a machine learning algorithm, this way obtaining the label for the patch as an output.

3.4 DPL Tool

The final step, after analyzing the patches and programming the SVM to classify them according to feasibility, is the creation of a tool capable of hot–patching the data structures of an application. The purpose of DPL is to address the conflicts created by modifying data structures with a patch. Therefore, DPL is used to update the data structures of an application according to the new semantics introduced by the patch.

If we go back to Figure 1.1, the DPL tool is capable of updating *variable* according to the value of the condition on the if case. DPL is integrated with three different libraries: Pin, MySQL, and DWARF, which are explained in Chapter 7. The DPL tool is implemented using four different procedures which are explained in the following subsections.

3.4.1 Pin Library

Pin is a dynamic binary instrumentation library that enables the creation of dynamic program analysis tools [37], we decided to work with Pin because of the advantages it offers

when instrumenting an application. Pin offers a rich, and easy to interact, API that let us inject arbitrary C and C++ code into a running application. Besides this, we can also get the state of the computer architecture the application is running on when attaching to the process.

3.4.2 Data Updating

DPL framework updates data by following three steps. The first step of DPL is to import all of the data from the application we want to patch. We call this the **import routine**. After the tool imports all of the variables by using the debugging information from the application, every data structure that is currently active on the stack is stored into a MySQL database.

After the import routine finishes, the data patch object is applied and everything is updated according to the patch. The data patch object consists of a set of SQL queries that aid DPL to update the data structures, and which are issued to the MySQL database for updating the stored variables.

After updating the data structures, everything is exported back to the application and updated accordingly. This process is called the **export routine** and it is the final step of a three steps process.

DPL framework works with two other APIs for data patching. The first API is the DWARF API, which offers the debugging information of the application in the form of DIEs (Debugging Info Entries). One limitation of DPL is that, if we want to patch an application, we need to have the debugging information. We can obtain it, however, by compiling the application with the -g flag on GCC.

The second API used by DPL framework is the MySQL C API. This API communicates our framework with the database on where every data structure is stored for being updated. The API queries the database for both the importation and exportation routines. More specifically, it let us store data into the database by using the **CREATE** and **INSERT INTO** commands, and it also let us retrieve data structures from the database by using the

SELECT FROM command and handling the results with a respective function. Another advantage that the MySQL C API offers is letting us modify variables from the database for the exportation routine. For example, if we want to modify a particular variable, we can use the **SET** command and modify the data of it. Then, by having the address of the variable as metadata, the DPL framework is able to export back the data to the address we want to modify.

3.4.3 Data Patch Object

If we go back to Figure 1.1, we can express this not only in primitive operations, but in a set of SQL statements that is capable of patching the data inside the database according to the patch.

A DPL object, with an SQL translation for the patch in Figure 1.1, would be as follows:

---

procedure DPL Translation of Patch1
SET data = 0 WHERE name = "variable" IF name = "condition" AND data > 10
end procedure

As we can see, we are able to interpret a security patch using 6 operations from our language as a data patch consisted of a single SQL statement. In order to apply the data patch, we first pass it as a parameter to DPL. Then, DPL executes the SQL statements by using the MySQL API over the database. After applying the data patch to the database, MySQL updates the variables inside the database according to the patch.

Now, for more complex patches, let us refer back to Figure 1.2 and Figure 1.3 from Chapter 1. We can see that in Figure 1.2, there are two major data modifications (i.e., The data structure being smb_off2 and first declared and then modified). By following our heuristics we can now translate these source code instructions into primitive operations, as such:

Declaration of smb_off2: **Create**(smb_off2)

Modification of smb_off2: **Search**(smb_off2) *then* **Modify**(smb_off2)

These statements refer to two different set of algebraic operations which we called DMMs. We explain the concept of DMMs in Chapter 4.

Chapter 4

Analysis of Patches

This chapter explains the details of our translation procedure from a set of C language statements to a graph description language. We then study the properties of the graph and find if there exists a relationship with feasibility. We can see this chapter as an empirical study of 75 different patches and as the ground truth for our experimentation in further chapters.

4.1 Intermediate Language for Machine Learning

Previous work on applying machine learning techniques in source code [59] takes a similar translation approach, and other graph techniques are also used for virus detection [33]. Since feature vectors are suitable as input to SVMs, we needed to translate the statements in source code 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.

Translating Patches into Graphs:

The process for translating security patches into graphs consists of the following steps:

1. Express statements that are modifying data structures using our set of relational–algebraic operations (i.e., search, write, read, compare).

2. Use the syntax presented in Table 4.1 to express a statement or operation modifying data as a node.

3. Connect the nodes according to the scope of the operation (e.g., an if case proceded by data modification would be a circle proceded by a dot).

4. At the end, the graph expresses every data modification inside a security patch. The leaves of the graphs are the data modification statements (i.e., operations 1, 2, 3, and 4 in Table 4.1).

After following these steps we have a graph that represents how data modification statements interact with each other inside a security patch. We use the attributes of these graphs (e.g., number of nodes, longest path) as part of our set of features for our SVM.

4.2 Our data–patching language

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 our language for translating patches into data patches and they are used as an intermediate language for producing the set of SQL statements. Each operator can be described in the following way:

1. **Search:** It is 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 have metadata such as the 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() [1].

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 passing 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 a variable and apply some heuristics on it. Most of the time, patches use read operations to compare variables to values or write values in 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.

4.3 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. The behavior

[1] 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.

of some functions are easy to predict by reading the manual page, as long as they are functions from the C library.

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 deduce 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 approach, such 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.

4.4 Data Modification Machines

We can define 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 • in graphs and as the operations *search and write* in our language. The computational cost of this operation is of 2. The computational cost is calculated by counting the number of operations it takes to achieve the task (i.e., search + write).

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 branches with data modifications inside.

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.

We can express a limited number of other operators that can potentially modify data structures. These operators include:

1. String Length: By getting the size of a character array. We are not able to deduce character pointers.

2. Addressof, typeof, sizeof: Metadata of each variable.

3. Free: By setting the value of the variable on a particular address to NULL.

4. Function returning values: For this operation, we assume that the function will return correctly.

4.5 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 4.1, 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 example in pseudocode

```
1: procedure DataPatch1
2:     int var = −1
3:     if var < 10 then
4:         if var2 < 10 then
5:             var2 = 8
6:         elsefinish
7:         end if
8:     elsefinish
9:     end if
10:    if var < 5 then
11:        if var2 < 2 then
12:            free(var2)
13:            var2 = 10
14:        elsefinish
15:        end if
16:    elsefinish
17:    end if
18: end procedure
```

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

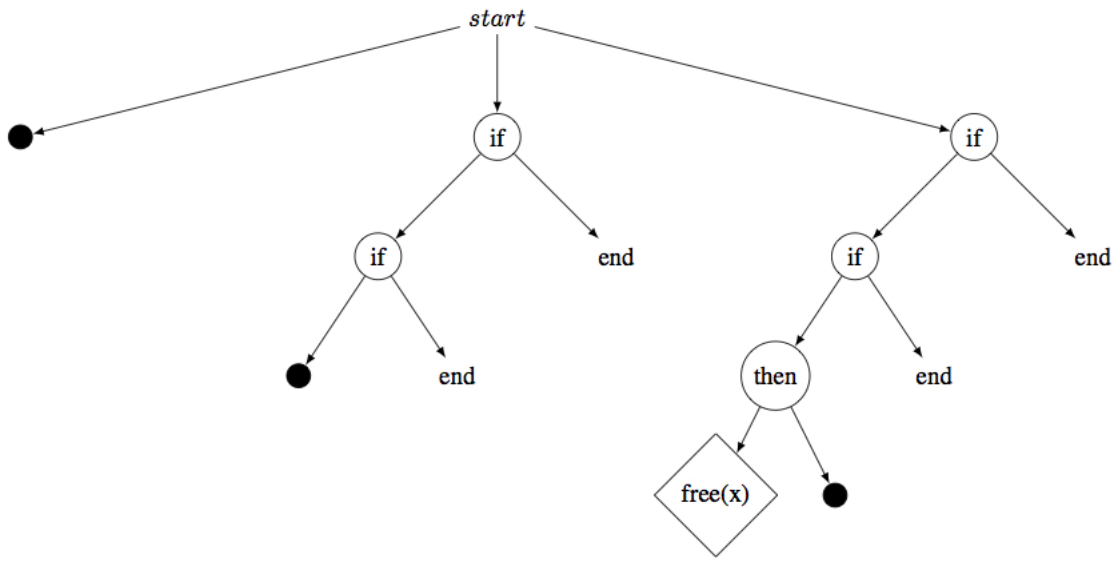


Figure 4.1: Graph representation of the control flow of data modifications for Algorithm 1.

In the Algorithm 1 example, the pseudocode expresses a patch that has four different data modifications. The statements 2, 5, 12, and 13 are modifying some variables in 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 4.1. 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 4.1). The combination of all these machines defines a data patch. We express the combination of these operations graphically in Figure 4.1.

We present another example of a different patch in Algorithm 2. In this example, we have

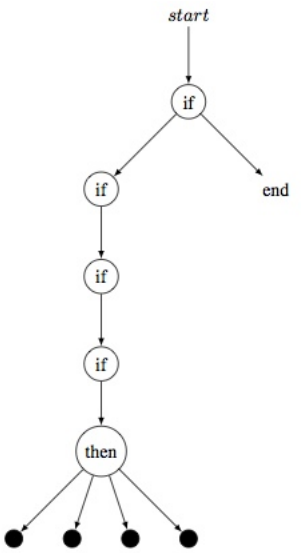


Figure 4.2: Graph representation of the control flow of data modifications for Algorithm 2.

four different data modifications over four different variables. Before the data modifications, however, there is an if case with three conjunctions. In our language this can be expressed as four different if cases and whenever all of them are met we will get to modify the variables. The Algorithm 2 is translated into its graph description language and expressed in Figure 4.2. As illustrated, there are four different leaves representing the data modifications, which are also four Machine#1 instances of Table 1.

Algorithm 2 Patch 2 example

```
1: procedure DataPatch2
2:     if var < 10 and var2 = 0 and var3 > 10 and var4 ≠ 0 then
3:         var1 = 1
4:         var2 = 2
5:         var3 = 3
6:         var4 = 4
7:     elsefinish
8:     end if
9: end procedure
```

4.5.1 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 4.1 and Figure 4.2 express two different patches and how data modification statements are defined. As illustrated in Table 4.1, the leaves of each graph are the data modification statements ($\bullet$ and $\diamond$). These statements represent some modification of a variable or data structure. If we refer back to Figure 4.1 we can notice that the leaves are the $\bullet$ representing the use of an assignment operator (e.g., var = 0) or the $\diamond$ 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 $\bullet$ know, based on the roots of the leaves, how to translate that statement into a data modification machine. As an example, if we refer to Figure 4.2 we can see that in order to modify data using the $\bullet$ 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 tells about the patches, that of the control flow of the patch, how data is being modified (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 any data.

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. [57]. 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.

4.5.2 Feature Vectors

In order to classify patches, we first define some of the attributes that we considered to be important in the graph representation from a patch. We analyze each graph and create a feature vector that consists of different attributes mapped to a feature space. The feature vector is a representation of the complexity of a patch according to the attributes of its graphical representation. This complexity informs the difficulty of translating a patch to a data patch. For a detailed analysis of the features, we can refer to Table 7.2.1

The features that we considered for each feature vector are: number of operations (e.g., 2 searches, 3 writes, 2 reads = 7 operations); node degree (i.e., number of nodes); longest path of the graph; number of searches, compares and writes; maximum number of inside and outside degrees for a node; number of operations in a cycle; functions that return values; functions on where parameters are passed; number of connected components (i.e., exit nodes); number of edges; cyclomatic complexity; and percentage of expressibility.[2] Cyclomatic complexity [28] is defined via the formula:

$$Cyclomatic\_Complexity = |E| - |V| + |N| \tag{4.1}$$

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

[2]We considered percentage of expressibility as the percentage of statements that we are able to express using primitive operations.

We define the feature vector for Algorithm 1 as:

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

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

The resulting vector has a number of features; this large group of features are seldom uncorrelated, and most classifiers work well with a smaller set of features or linear combination of features. This challenge raises the question of how to choose the most significant set of features. As detailed in the next section, we applied PCA over the set of features to obtain the most independent features.

4.6 Summary

The purpose of the manual analysis of patches is to understand how patches modify data by studying the statements that assign values to data structures. We were able to categorize different statements according to how they modify data and mapped them to basic database operations such as: read, write, search and compare. We also translated the C source code of a patch into an intermediate graphical representation that illustrates the control flow of the patch; the leaves of the graph are data modification statements, and the subtrees represent the control flow of how the patch manages the statements.

Table 4.1: Data modifications machines (DMMs) are described in this Table. We define a DMM to be a group of statements that, by interacting with each other, modify data structures.

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 4.2: Different types of features for our feature vector.

Chapter 5

Applying SVMs

This chapter contains an exposition of our experimental methods and how we use 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 suitable or unsuitable candidate to hot patch. Classifying source code is not an easy task and there are many options (neural networks, bayesian classifiers, genetic algorithms, etc. [32]). However, we decided to use SVMs because of the related work we described in Chapter 2 on classifying source code with machine learning techniques.

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. In this chapter we present the methodology behind our SVM classification.

5.1 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 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.

5.1.1 Standardizing Data

Each feature vector has many attributes that are correlated with each other, this is the type of data that PCA takes as input. PCA gets possible correlated variables as input and maps them to a smaller dimensional subspace. In this way, PCA decides which features are the most independent and reduces our dimensionality to be less than or equal to the current number of features, in this case fourteen. However, before applying PCA we must first normalize 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 normalize the values of our dataset to be between {-1, +1} using this formula:

$$\frac{2(x)}{max(x)} - 1 \tag{5.1}$$

In the previous formula, the denominator $max(x)$ is an integer value larger than zero.

5.1.2 Applying PCA

We applied PCA using a free software programming language called R[31] that is used for statistical computing. More specifically, 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 5.1, there is a large correlation between the features: number of writes, number of searches, computational 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 5.1, 5.2, and 5.3 – 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 $\overrightarrow{PCA\_FV}$[1] as follows:

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

can be mapped to:

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

5.1.3 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. 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.

[1]Data, in this example, is not normalized.

We first train our machine using the **ksvm** function in R and our training dataset. We used five different kernel functions – for the kernel trick [53] in our SVM – to get different results when using our SVM. These functions were:

- Linear kernel: it is given by the inner product $< x, y >$ plus a constant. Its formula is:

$$k(x, y) = x^T y + c$$

- Polynomial kernel: This is a common function used when data is normalized (like in our case), its formula is:

$$k(x, y) = (\alpha x^T y + c)^d$$

- Radial Basis kernel: This function can be adjusted using a parameter *sigma*. In this case we used the one suggested by R, but we can also tune this parameter and get different results. Its formula is:

$$k(x, y) = exp(-\sigma \|x - y\|^2)$$

- Anova kernel: It is a type of radial basis function that performs in multidimensional regression problems. Its formula is:

$$k(x, y) = \sum_{k}^{n} = 1exp(-\sigma(x^k - y^k)^2)^d$$

- Laplacian kernel: It is another type of radial basis function, however it is one of the less sensitive function for changes in the sigma value. Its formula is:

$$k(x, y) = exp(-\frac{\|x - y\|}{\sigma})$$

We train the SVM using the R programming language [31] with the commands shown below. We first import data from a 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
```

We have to remember that data is classified according to how we trained the SVM, thus the results may vary depending on how many data points and which features we use for training. We decided to divide our dataset into two parts: the training and the testing dataset. As explained before, the training dataset was first normalized and finally analyzed by PCA. After this analysis, we get a gram matrix representing that works as input for our SVM.

5.2 Summary

At the end of the last chapter, we were able to get a feature vector that represents a measurement of our definition of complexity of a patch. However, because we wanted to be as broad as possible when choosing the features, we needed to apply a dimension-reduction

technique in order to map the 15-Dimensional Feature Vector into a lower dimension that made the process of applying a learning algorithm easier to achieve. We also applied a statistical technique called Principal Component Analysis (PCA) to reduce the dimension of the vector and then use the new vector as input to our SVM. The results for the SVM classification are presented in Chapter 7.

In the next Chapter, we are going to implement our DPL framework. we designed and implemented a proof of concept for DPL to hot–patch different statements encountered in our analysis of security patches.

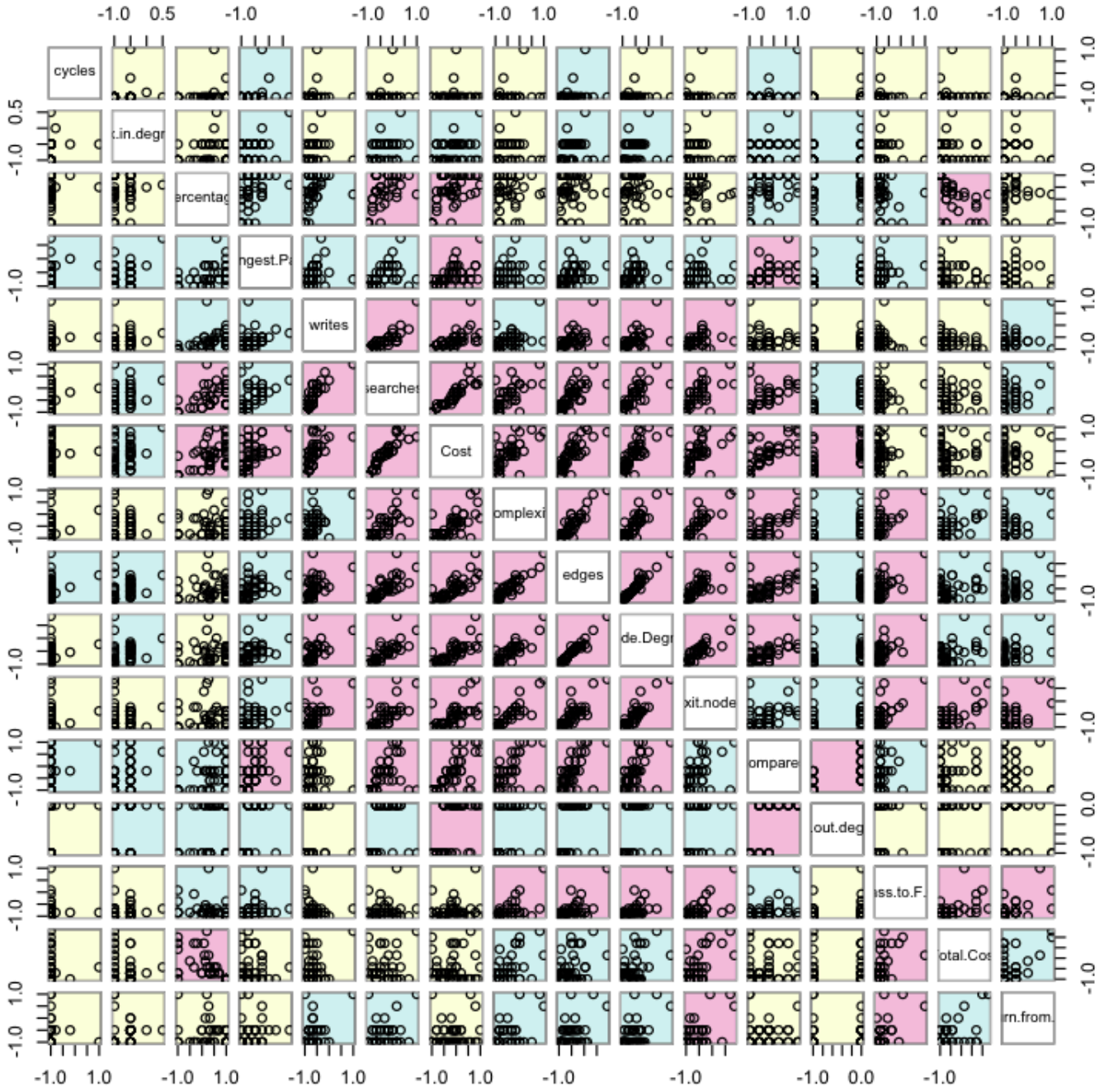


Figure 5.1: This figure illustrates the correlation of all the features. The dark colors indicate more correlation. In this case we only considered the pink squares as the correlated variables, making the first four features the most independent and the ones to consider as the final features.

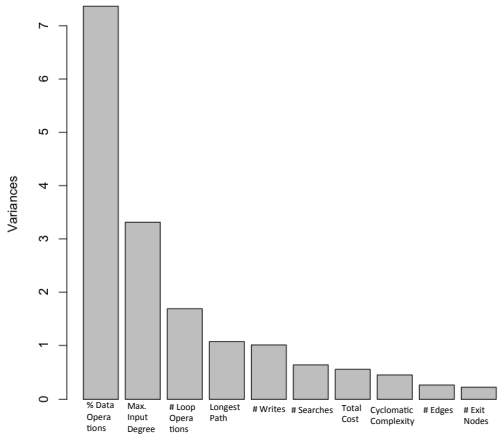


Figure 5.2: Variances of the features. According to Kaiser's criterion[67], we should only consider the components which variances are over 1, which are the first four features.

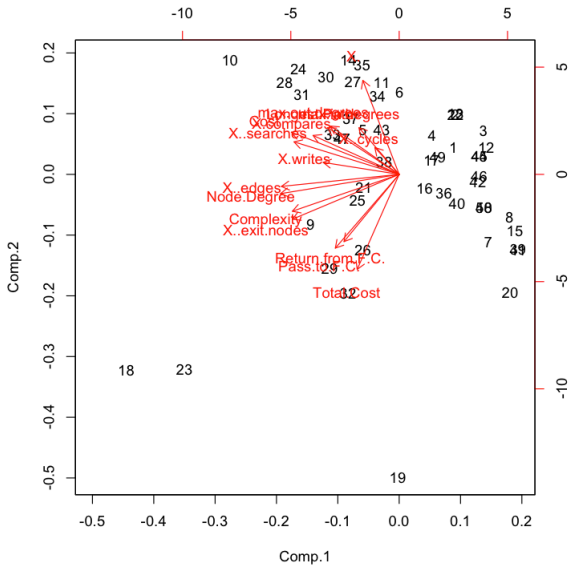


Figure 5.3: The figure shows the correlation of all the features as vectors. The closer the vectors are, the more correlated.

Chapter 6

DPL: A Data Patching Language System

In this chapter, we describe DPL, a system for hot–patching data structures on an application according to a security patch. The system was designed as a proof of concept for augmenting the validity of the empirical studies presented in Chapter 4. The implementation of DPL consists of over 2500 lines of code as a Pin tool [37] and is integrated with the MySQL and DWARF APIs. It also consists of three main stages that are explained in this chapter and was used to experiment with patching (i.e., applying security patches) or modifying data structures (e.g., altering a linked list, adding or eliminating nodes of a data structure).

Main Objective of DPL

The main objective of DPL is, by passing an object in the form of a data patch to a running process, to update the data structures of a running application according to that data patch. After attaching to the process, this entails DPL to use the data patch for updating the data structures of the application in order to be consistent with the new data semantics the patch introduces to the application.

6.1 Design Overview

We can see DPL as a three–step mechanism. The first step consists of the import of every data structure in the application into a database. The second, consists of the application of the data patch object or **the execution of SQL statements, result from the translation of a patch into our language, over the database**. The third and final step is the export of the modified tables, representing different types of data structures, back to the application.

In this section we explain how the three steps of DPL interact with each other. The

application of the data–patching stage is briefly explained here and can be complemented with Section 6.3.

6.1.1 Import Routine

As stated before, our mechanism to update data structures is using a relational–algebraic approach language following the principles of database operations. We decided to use SQL as a query language to update the data structures from the application because it offers a rich enough language to express most of the data modification statements. One advantage that working with SQL offers is that we are able to expand its language by using a procedural query language, therefore expanding our set of operators.

The import mechanism consists of the analysis of a **memory image** of the application that we want to patch to import every instance of data structure types into a MySQL database. We then stored these data structures into different tables according to their type.

As Figure 6.1 expresses, we separate data structures according to their scope as global, local or dynamically allocated, the latter being mostly pointers that pointed to a particular data structure. We recursively recovered the dynamically allocated structures by using the pointer as the root of the structure and, after getting the members of the structure, use the size of each member to recover their values.

Parsing these pointers and extracting the data structures to store them into the database was the most challenging task of the implementation of DPL and is related to the concepts applied by shape analysis techniques [16, 17] explained in Chapter 2.

6.1.2 Patching data structures

Applying a data patch consists of a translation of the data semantics of a security patch into a set of SQL statements and the metadata of the variables we want to patch. For our proof of concept, we also included the address of the function that is *patching* the previous function, but the patching of source code is going to be automated in future work. DPL applies the

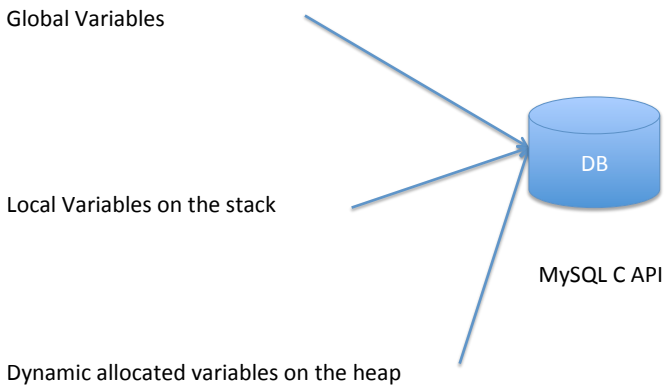


Figure 6.1: This figure illustrates the three different types of variables that get stored into the MySQL database by DPL. After being stored, we run the DPL data patch – or set of SQL statements – over the database in order to update them.

data patch object **after** importing every data structure into the database. **We can see a data patch as the combination of different DMMs for translating a patch**.

6.1.3 Export Routine

DPL exports and updates only the tables that were modified by the data patch object. The data patch object contains a set of statements in SQL that includes the tables we want to update. After updating the tables, we can extract the values and addresses that were updated from the database. Then, we can reference the address and assign a value by using the advantages C as a programming language offers to our system (e.g., allocation of memory, pointer referencing and dereferencing). The data structures are then updated according to the data patch. An example of how we assign a new value to a data structure is:

```
* address = new_value;
```

Metadata for deducing data structures

In order to be able to patch data structures in the running application, we need some metadata from each data structure. This helps both the import and export mechanisms to identify easily the data structures we want to update. The elements that we consider as metadata for updating data structures are:

- Address: In order to update the value of a variable we need its starting address. Because we are working with the C language, we can dereference an address to get the value stored on it. It is also possible to update the value inside the address using the unary operator $*$. By doing a $*address = new\_value$ we are able to assign a new value for the data structure.

- Data: It is important to know the value of the data stored in a variable. Sometimes we need to also compare variables in order to patch them (e.g., when we have an if case proceded by a data modification). The data is stored

as a hexadecimal values in our database and can denote a character or a number (float, double or int).

- Name: When modifying a data structure, we need to have a character array with the name of the variable. This is stored into the database and then, when issuing a **modify** or **read** statement, can be used to look for a particular variable.

- Byte Size: To be able to **modify** a variable, we also need to know the size of the variable to know its ending address (by adding the size to its starting address). The DWARF info offers the size for the data structures declared in the application, and if we have a pointer to a data structure we are able to dereference the pointer by using the DWARF info of the structure as a guide. However, character pointers were treated as single characters because the DWARF info does not offer its size.

For the purposes of data patching, we decided to store variables in our database according to their scope, and type. It was also easier to parse the DWARF info according to the scope of the variable, because it is organized by subprogram (i.e., function).

6.2 Storing data structures according to their type

DPL separates data into two different groups: according to their scope, or to their type. If the data structure we want to update is a primitive data type (i.e., int, float, double, char, or void) the structure gets stored according to its scope. If the data structure is of a more complex type (i.e., declared as a *struct* in C), then it gets stored according to its type. Arrays or lists are also considered to be of primitive types and are stored according to their scope.

In this section we explain the two different classifications we do for separating data structures. The first one with primitive types that are separated according to the scope

```
struct STRUCTURE2{
    int y, h;
    float f;
    char * r;
    struct STRUCTURE1 * tob;
};

int global_variable = 10;
struct STRUCTURE2 global_structure;
```

Figure 6.2: Example of a variable that is considered to be global when storing it in the database. The variables *global_variable* and *global_structure* are considered to be global because they are not declared within the scope of a function. DPL stores global_variable into a table called GLOBAL_VAR, and global_structure into STRUCTURES.

of their declaration. The second classification with more complex types that are separated according to the semantics of the structures (e.g., a linked list is stored in a table named as the type of the list). DPL classifies this information after parsing the DWARF information of the application.

6.2.1 Primitive types

Data structures declared as primitive types are classified according to their **scope**. By using the DWARF info, we are able to deduce if a data structure is of a primitive type based on the offset, inside the DWARF info, of its type. In Chapter 2, we explained how data types are declared in the DWARF info and which offsets are related to primitive types. In this section, we explain when variables are considered to be global or local and how the DPL system classifies them.

Global Variables

Global variables are those variables that are declared outside the scope of any function. In Figure 6.2, we can see an example of what we consider to be global variables in C when importing data to the database.

Global variables, with primitive types, are treated independently by DPL framework and put together in a single table called "GLOBAL_VARS". On the other hand, more complex data structures, declared as global variables, are treated as structures and put into a different

table according to their type.

If we refer back to Figure 6.2, we can see that there are two global variables being declared. The first one is *global_variable*, a variable of type **int** and with a value of 10. The second variable is an instance of a data structure of type *STRUCTURE2*, with a name of *global_structure* and with no data stored inside it.

If we refer to the table below, we can see how global variables are stored in the database. This table contains every global variable declared as a primitive type (e.g., int, char, float, and double). In this case, we are only working with one global variable with a primitive type *global_variable*. Its metadata consists of a type int, a size of 4 bytes, with an address of 0x804a028 (which is within the range of global variable addresses 0x804...), and a decimal value of 10.

ID	name	type_name	type	bytesize	address	value
1591	global_variable	int	DW_TAG_base_type	4	134520872	10

Local Variables

Local variables are defined as the variables that are declared in the scope of a function. In Figure 6.6, we can see an example of how local variables can be declared in the C language. In the case of local variables, we patch them as long as they are currently active on the stack. When attaching DPL to a running program, we extract every local variable, with a primitive type, that is currently active on the stack and store them into the "LOCAL_VAR" table.

The DWARF debugging standard offers information about where the local variables are located in the stack by giving the position of the variable inside the stack. As an example, a DIE for a variable declared as local inside the application is illustrated in Figure 6.4. The DW_AT_location tells that the variable is local and that we need to add 44 bytes to the register *breg*4 (*esp* in x86) for getting the address where that local variable is stored. After

```
address
get_address_withsize(int size, Dwarf_Die print_me){
    address return_addr;
    Dwarf_Error error;
    Dwarf_Loc * dwarf_location;
    Dwarf_Attribute * atlist;
    Dwarf_Signed atcnt;
    Dwarf_Signed lcnt;
    Dwarf_Locdesc *llbuf = 0;

    res = dwarf_attrlist(print_me, &atlist, &atcnt, &error);
    res = dwarf_loclist(atlist[atcnt - 1], &llbuf, &lcnt, &error);
    if (res == DW_DLV_OK) {
        dwarf_location = llbuf -> ld_s;
        return_addr.addr = (unsigned int) (dwarf_location -> lr_number) + size;
        return_addr.num = (int)(dwarf_location -> lr_atom);
    }

    return return_addr;
}
```

Figure 6.3: This particular function returns the address of a particular DIE by parsing the DWARF info of the application.

```
<2>< 1214>  DW_TAG_variable
    DW_AT_name                  mts
    DW_AT_decl_line             69
    DW_AT_type                  <90>
    DW_AT_location              DW_OP_breg4+44
```

Figure 6.4: Example of a Debugging Information Entry (DIE) of a variable called *mts*. AT_name contains the name of the variable, AT_decl_line the line where it was declared, AT_type the DIE position for its type (90 is the position for integer types), and the AT_location which specifies the position of the variable inside the stack.

dereferencing that address, we are able to get the value for that particular variable, and we can use the metadata that the DIE offers for storing the address, type, name, and size of that variable. We used the function presented in Figure 6.5 to get the value of a size by passing its address and size. To get the address of local variables we use the function illustrated in Figure 6.3.

name	offset	bytesize	data	address
length	-36	4	7	3217882940
struc_	-28	4	0	3217882948
top	-24	4	0	3217882952
instan	-44	8	30705	3217882932
i	-20	4	10	3217882956
cms	4	4	3	3217882980
mts	0	4	4	3217882976
jos	-28	4	1110882386	3217882996
tiburc	-24	4	1119917179	3217883000

```
//The address + offset and the size of a DWORD
unsigned int
EBP_data(ADDRINT EBP, int size){

   std::stringstream ss;
   unsigned char contents_address = 0;
   unsigned char * address = (unsigned char *) EBP;
   unsigned int m = 0;

   for(int i = size - 1; i >= 0; i--){
       contents_address = *(address + i);
       m += contents_address * pow(256, i);

       ss << hexifyChar(contents_address);
   }

   const std::string& temp = ss.str();
   const char * cstr = temp.c_str();

   printf("(DATA_PARSING) %08x -> %s\n", m, cstr);

   return m;
}
```

Figure 6.5: We use this function for dereferencing and getting the value of a particular variable by passing its address and size.

```
void add(int mts, int rts){
    int length;
    float tiburcio = 96.29;
    float jos = 45.68;

    length = mts + rts;

    subs(4, 3);

    return_add(9, 5);

    printf("%d\n", length);

    printf("%d -> %u\n", rts, &rts);

}
```

Figure 6.6: Example of variables that are considered to be *local* when storing them in the database. The local variables in this example are *length, tiburcio,* and *jos*; *mts,* and *rts* are parameters of the function and are also in the scope of the add() function. Even though technically they are not local variables they are considered as local variables by DPL.

```
struct localvars{
   Dwarf_Die prim_die;
   unsigned int offset, low_pc, type_offset;
   int size;
   int type_tag, pointer_tag, num_ptrs;
   char * local_name;
   struct localvars * next;
};
```

Figure 6.7: This figure illustrates a structure we declared for storing all of the local variables we get from the stack. We then use this structure for storing the local variables of the application in our database.

```
| length | -20    | 4        | 20         | 3217883004 |
| rts    | 4      | 4        | 10         | 3217883028 |
| mts    | 0      | 4        | 10         | 3217883024 |
+--------+--------+----------+------------+------------+
```

6.2.2 More complex types of structures

We define a *complex* type of structure to be a structure declared with a type different than a primitive type. These types are declared as a structure using the **struct** type in C and sometimes can be defined as a type by using the **typedef** operator. Their purpose is to construct more complex structures such as linked lists, trees, hash tables, or many other types.

For explaining the data structure recovery (DSR) process, we work with two different structures. The first structure is of type *STRUCTURE2* from Figure 6.2, and the second structure is of type *LINKED* from Figure 6.10. The variable of type *LINKED* is *struct_li*, which is a dynamically allocated linked list with two members.

After attaching to the process and extracting every variable and data structure from the program, DSR rebuilds the active data structures by using our set of heuristics over the database. We can identify complex data structures on the application by using the DWARF info. The DWARF info uses a pre-determined offset for primitive types (e.g., 90 for int, and 162 for char) and nests a complex structure according to its type. For example, if we want to recover a *LINKED* structure, then we will have to look for the entry of *struct_li* in the DWARF info and its type will be pointing to the offset of *LINKED*. Then, the children of the type *LINKED* will be its members (i.e., *y, p* and *next*) and the members will have a primitive type of int[1]. More precisely, we can summarize this process as:

1. The first feature the DSR process looks for is which data structures are declared as *struct*. It also looks for which structures were defined by using the *typedef struct* operator in C. All variables declared using these operators are considered

[1]Except for next, which is a pointer to another structure of type *LINKED*.

to be structs. To detect if a data structure is not of a primitive type, and thus a more complex type, we use the function illustrated in Figure 6.9. Then, if the data structure is a more complex type, we use the function in Figure 6.8 to get its type.

2. Next, DSR separates structures from pointers-to-structures. This is done by parsing the DWARF info and looking for the $\widehat{\text{DW\_AT\_PTR}}$ attribute. Whenever DSR finds a pointer to a structure, it stores the pointer separately and parses the structure according to its type. In the linked list example, the pointer to a structure would be *next*.

3. After having each structure defined as a data structure, we need to store them into the database. Since every structure has different members, we need a table for each type of structure. We cannot store these structures according to their scope because we need to specify the type and value of each member of the structure.

4. The first step for storing the structures into the DB is to store the instances of each structure into an *INSTANCES* table of the DB. The *INSTANCES* table has every instance of any data structure declared in the program, the type of the structure is represented with its DWARF info attribute offset and its name. In the case of the type *LINKED*, its instance is *struct_li* and its type *LINKED*.

5. We also need to parse the members of the data structures and store them into a different table. This table is called MEMBERS and it has stored the members of every instance of a data structure type. In the case of the type *LINKED*, it has 3 members and the number of instances depends on the number of elements on the linked list (i.e., how many times a new structure with type

```
Dwarf_Die
get_structure_type(Dwarf_Die return_sibling, Dwarf_Debug dbg){
    Dwarf_Attribute *    atlist;
    Dwarf_Signed         atcnt;
    Dwarf_Off            offset = 0;
    Dwarf_Die            print_me = NULL;
    int                  res2;

    res = dwarf_attrlist(return_sibling, &atlist, &atcnt, &error);
    if(res == DW_DLV_OK){
        for(int i = 0; i < atcnt; ++i){
            res2 = dwarf_global_formref(atlist[i], &offset, &error);
            if(res2 == DW_DLV_OK){
                res = dwarf_offdie(dbg, offset, &print_me, &error);
            }
            dwarf_dealloc(dbg, atlist[i], DW_DLA_ATTR);
        }
        dwarf_dealloc(dbg, atlist, DW_DLA_LIST);
    }
    return print_me;
}
```

Figure 6.8: The figure illustrates how to get the type of a structure by giving its DIE. The DWARF info has a nesting according to how the type is declared.

LINKED gets allocated).

6. If we would like to modify the members from the data structure *global_structure* from Figure 6.2, we would need to modify the addresses for each member separately. DPL integrates with DSR by querying the DB and modifying the addresses according to the size of the member.

6.3 Data Patch Object

A data patch object is a plain-text file consisted of two sections. The first section relates to the patching of source code, it consists of a header for including the address of the function we want to patch. The second section consists of a set of SQL statements that are issued to the database in order to update the data structures encountered in the application. An example of a data patch object is illustrated in Figure 6.11.

How to translate source code into a data patch object?

The translation consists of parsing the set of SQL queries, send them to the database and update the values of the data structures that are being modified by the patch according

```
int
isStructure(Dwarf_Debug dbg, Dwarf_Off offset){
    Dwarf_Die      print_me;
    Dwarf_Error    error;
    int            result;
    Dwarf_Half     tag;
    const char *   tagname = 0;
    char *         name = 0;

    res = dwarf_offdie(dbg, offset, &print_me, &error);

    res = dwarf_tag(print_me, &tag, &error);
    if(res != DW_DLV_OK) {
        printf("Error in dwarf_tag");
        exit(1);
    }
    res = dwarf_get_TAG_name(tag, &tagname);
    if(res != DW_DLV_OK) {
        printf("Error in dwarf_get_TAG_name");
        exit(1);
    }
    res = dwarf_diename(print_me,&name,&error);
    if(res != DW_DLV_OK){
        return 0;
    }
    result = strncmp(tagname, "DW_TAG_structure_type", 100);
    if(result == 0)
        return 1;
    else
        return 0;
}
```

Figure 6.9: This figure illustrates a function that returns a true or false depending if the DIE being passed if a structure or not.

```
struct LINKED * struc_li = NULL, * st_head = NULL;

for(i = 1; i < 10; i++){
    struc_li = (struct LINKED *) malloc(sizeof(struct LINKED));
    struc_li -> y = i;
    struc_li -> p = 12;
    struc_li -> next = st_head;
    st_head = struc_li;
}
```

Figure 6.10: An example of a dynamically allocated linked list in C. The members of the structure are *y*, which values depends on the variable *i*, *p*, which value is of 12, and *next*, which is a pointers that points to the next element of the linked list.

```
SQL STATEMENT:
Here we introduce the SQL statements we want to issue
according to a security patch.

VARIABLES:
Name of the variables we want to modify.

TABLES:
Tables were we can find these variables in our database.
```

Figure 6.11: This figure illustrates how to construct a data patch object. In Chapter 7 we use this model for constructing a data patch object for a C application, according to the modifications we want to issue.

to the queries.

We can construct a data patch object of the security patch example in Figure 6.12. For this, let us answer the following questions: How does the patch modify data structures? What can we learn about the patch in terms of data modifications?

- Identify the data modification statements: According to the analysis of our data corpus, we are able to identify over 60% of our security patches modify data structures. To identify data modification statements, we should look for assignment operators (e.g., $=, +=, -=, *=$).

- Identify any other data modification statements (e.g., functions): There are many ways to modify data structures other than using assignment operators. One example is the $free()$ function, commonly used to free a variable that was previously allocated.

- Understand the operations involved in the data modification: It is common to have a data modification expressed as a set of statements. Sometimes assignment operations are dependent on other operations like comparisons (e.g., if cases conditions), returned values from functions, etc.

- Select the proper data modification machine to patch the statement: If we look

back at Table 4.1 from Chapter 4, we can match a data modification machine with a statement – or set of statements – that we want to patch.

The purpose of this section is to understand how the procedure for creating a data patch works and how we can translate C source code into SQL statements. The translation process of a patch into a data patch object is as follows:

1. The first step is to gather all of the statements that the patch updates. If we refer to Figure 6.12 we can conclude that the statements can be expressed in pseudocode as:

   ```
   if an element of a structure is equal to APR_SUCCESS then:
   modify the element to be equal to APR_GENERAL;
   ```

2. Next, it is crucial to know which statements are modifying data and how we can express them in primitive operations. This translation, in pseudocode, is as follows:

   ```
   search for element
   compare it to 1, if it is equal then
   modify element to be 0;
   ```

3. Now, because our framework is communicating with a database, we need to translate the statements to SQL syntax. This operations can be translated into the following SQL syntax:

   ```
   SELECT member = rx FROM inctx IF rx.data = 1
   THEN SET VALUE rx.data = 0;
   ```

```
RCS file: /home/cvspublic/httpd-2.0/modules/ssl/ssl_engine_io.c,v
retrieving revision 1.100.2.14
retrieving revision 1.100.2.15
diff -u -r1.100.2.14 -r1.100.2.15
--- httpd-2.0/modules/ssl/ssl_engine_io.c       2004/04/15 19:41:39     1.100.2.14
+++ httpd-2.0/modules/ssl/ssl_engine_io.c       2004/08/23 14:59:52     1.100.2.15
@@ -587,6 +587,10 @@
     while (1) {

         if (!inctx->filter_ctx->pssl) {
+            /* Ensure a non-zero error code is returned */
+            if (inctx->rc == APR_SUCCESS) {
+                inctx->rc = APR_EGENERAL;
+            }
             break;
         }
```

Figure 6.12: Example of a patch for Apache HTTPD. The patch modifies 4 different statements of the application, however only two of them are related to data structures. It is worth mentioning that in this case there is an additional if case that is also related to the data semantics of the patch (i.e., the first if case, declared before the patch modifications). We can express the two statements modifying data by using Machine#5 of Table 4.1 on Chapter 4.

6.4 Implementation

DPL uses a dynamic binary instrumentation tool framework called Pin that allows the creation of tools for the binary instrumentation of a program. DPL is a Pin tool consisted of around 2500 lines of code (including white spaces and comments). We implemented DPL in a 32-bit architecture with Ubuntu as an OS.

DPL analyzes the routines and instructions of the running application and executes the heuristics explained in this chapter by injecting its respective code. This code consists of the update of data structures following three main stages: import routine, data structure updates, and export routine.

6.4.1 Pin: A Dynamic Binary Instrumentation Tool

Pin lets developers create tools for injecting arbitrary C/C++ code in an application after stopping the process on a particular moment. The advantage of using Pin is that, in that particular moment, we can inject binary code into the application. Pin follows the principles of forensic analysis by letting us analyze the context of the application in the moment we stopped it. After analyzing the context of the application and injecting the binary code, Pin detaches from the process and continues its execution.

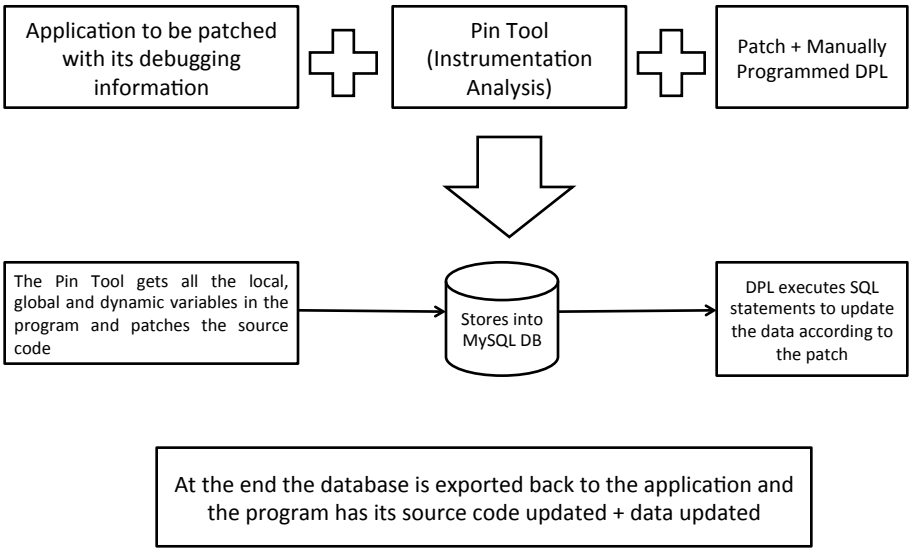


Figure 6.13: Diagram explaining how the DPL tool works, the process consists of three main stages: the importation routine that stores every data structure into a SQL database, the data patch that modifies the data structures by querying the database and the exportation routine that updates every data structure, according to the changes of the data patch, to the application.

```
VOID
SETRTN_CONTEXT(string * rname, CONTEXT * ctxt)
{
    registers reg;

    reg.esp = (ADDRINT)PIN_GetContextReg(ctxt, reg_esp);
    reg.eip = (ADDRINT)PIN_GetContextReg(ctxt, reg_eip);
    reg.ebp = (ADDRINT)PIN_GetContextReg(ctxt, reg_ebp);

    Get_LocalVariables(reg);

    PIN_RemoveFiniFunctions();
    PIN_Detach();
}
```

Figure 6.14: This figure illustrates how we get the context of the registers *eip*, *esp*, and *ebp* when using the DPL framework. This routine gets called by the function illustrated in Figure 6.16

Attaching to a running process is done via Pin by using the function *PIN_Attach(PID)* from the Pin API Library. The Pin API offers a communication between the DPL framework and the running process. We are able to get the state of the computer architecture when running the process that we are analyzing. Pin allows getting the context of the x86 registers and thus the local variables by using the *ebp* and *esp* registers as a guide. Besides getting the context of the CPU's architecture, we can also inject new code in the application. The new code is developed for the import and export of data structures to a MySQL database. One of the main advantages of Pin is that we can do all of this without restarting the running process.

The first step to run the DPL framework is to attach to the running application we want to patch. For attaching to a running process, we run the following command by calling our DPL framework:

./pin -pid NUMBER -t DPL.so

Where, *./pin* is the Pin tool library, the $-pid$ argument gets a process ID number as an input, and the $-t$ argument gets the Pin tool (of type .so) we want to use as an input, which is our DPL framework tool.

To illustrate the process of how the communication between our DPL framework and the Pin API works, we can refer to Figure 6.14, and Figure 6.15.

```
PIN_InitSymbols();
unsigned int * addr;

// Initialize pin
if (PIN_Init(argc, argv)) return Usage();

DPL = fopen("DPL.txt", "r");
if(!DPL)
    printf("File does not exist\n");

while(fgets(line, 80, DPL) != NULL)
{
     /* get a line, up to 80 chars from fr.  done if NULL */
     sscanf (line, "%d", &elapsed_seconds[i]);
     /* convert the string to a long int */
     i++;
}

addr = (unsigned int *) elapsed_seconds[0];
*addr = 2292;
conn = mysql_init(NULL);
pid = PIN_GetPid();
if(conn == NULL){
    printf("ERROR %u: %s\n", mysql_errno(conn), mysql_error(conn));
}

SetupRegisters();
```

Figure 6.15: This is the main function for our Pin tool. It initializes the Pin API, reads the data patch object and then sets up the registers to begin the import routine.

```
VOID
MemoryImporterExporter(RTN rtn, VOID *v)
{
   string * rname;
   RTN_Open(rtn);
   rname = new string(RTN_Name(rtn));
   for(INS ins = RTN_InsHead(rtn); INS_Valid(ins); ins = INS_Next(ins))
   {
       RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)SETRTN_CONTEXT, IARG_PTR, rname, IARG_CONST_CONTEXT, IARG_END);
       break;
   }
   RTN_Close(rtn);
}
```

Figure 6.16: When this function gets called, it initializes a routine for analyzing the context of the architecture DPL is running on. This routine is capable of setting the routines that get the context of the registers for getting the local variables of the application.

6.4.2 DWARF info

The debugging information of an application can be parsed by using a standardized format called DWARF. This format offers the debugging information of every function and data structure inside the program, including: the offset of each variable from the registers EBP and ESP; the name, byte size, address (location) of each variable; and the types that were declared in the program.

We can see the DWARF info as a summary of the context of the application, including its subprograms and variables. Each variable, type and functions are represented in the form of an entry called debug information entry (DIE), and each DIE has a number of attributes that represent the metadata of a variable. The type of a particular variable is an attribute of its DIE and it is expressed as its offset inside the DWARF info. For example, if we refer to Figure 6.17, we see that the first DIE has an offset of $\langle 1240 \rangle$. The list of attributes, that were used in DPL, can be described as such:

- DW_TAG_formal_parameter: Variables that are tagged with this attribute are parameters of a function.
- DW_TAG_variable: A variable was declared when its DIE is tagged with this attribute.
- DW_AT_name: This attribute refers to the name of the variable.
- DW_AT_type: This attribute refers to the type of the variable. In the case of the DIE in offset 1240, its type points to a different DIE. However, the DIEs that are declared as primitive types (e.g., int, char, float) point directly to their respective DIE type.
- DW_AT_location: Refers to the address of the variable. In this case, because the variable was locally declared, DW_OP_fbreg 4 means that the variable is 4 spaces above the address of the register ESP in the stack.

```
<2>< 1240>      DW_TAG_formal_parameter
                DW_AT_name                  mts
                DW_AT_decl_file             1 /home/robingonzalez/Downloads/pin/source/tools/MEMIE/Test Suite/Test3.c
                DW_AT_decl_line             77
                DW_AT_type                  <90>
                DW_AT_location              DW_OP_fbreg 0
<2>< 1254>      DW_TAG_formal_parameter
                DW_AT_name                  cms
                DW_AT_decl_file             1 /home/robingonzalez/Downloads/pin/source/tools/MEMIE/Test Suite/Test3.c
                DW_AT_decl_line             77
                DW_AT_type                  <90>
                DW_AT_location              DW_OP_fbreg 4
<2>< 1268>      DW_TAG_variable
                DW_AT_name                  length
                DW_AT_decl_file             1 /home/robingonzalez/Downloads/pin/source/tools/MEMIE/Test Suite/Test3.c
                DW_AT_decl_line             78
                DW_AT_type                  <90>
                DW_AT_location              DW_OP_fbreg -12
```

Figure 6.17: Example of a set of DIE frames inside the DWARF information.

Getting variables using DWARF:

When using DW_AT_location we can encounter several cases that are divided according to the scope of the variable inside the program. The three types of variables that we encountered in DWARF while using our test suite are:

- Global variables: We can get the virtual address of a global variable **directly** by getting its DW_AT_location, which will return an address in the global address space (i.e., an address located after 0x8040000).

- Local variables defined in the main function: These variables have follow different heuristics than regular variables because they have a DW_AT_location offset instead of an address. The offset is defined by having a DW_OP_breg4 + *offset* tag. The DW_OP_breg4 is the *ebp* register in 32-bit architectures and its offset is how many bytes away are the variables from EBP on the stack. By using a function that goes through the stack, we are able to get the address of a variable by using this information.

- Local variables defined in other functions: Similarly to the previous types of variables, we can get the address of one of these variables by getting its location tag and using it as an offset. In this case, these variables return a location

type of DW_OP_fbreg + *offset*, which can be translated to *esp* plus an offset.

We followed a different procedure for getting the dynamically allocated variables that is more related to pointer arithmetic. When a structure type was found, we used a lookup routine to find the type of its members. If a tag of DW_TAG_pointer_type was found as the type of one of its members, then we used recursion over this type to *rediscover* the structure.

Challenges of DWARF:

Most issues that we had while working with DWARF were mostly related to the way the API is documented. Here are a list of challenges that were found while working with DWARF:

- How to get the size of an array: The mechanism that DWARF uses for defining the size of an array of elements is different from getting the byte size of a variable. Figure 6.18 illustrates how we can obtain the size of an array when using DWARF.

- What most DW_OP_regs mean in a particular architecture: Another issue we encountered while parsing the DWARF info is the DW_OP declarations as DW_AT_location types. This field tells how the registers were declared according to the architecture of our machines.

- The way DWARF handles types of data structures took some time to understand because they are nested inside the DWARF info according to the type of the structure.

6.4.3 Patching the source code

Because the main focus of DPL is patching data structures, we simplified the task of hot–patching source code by following the technique from Miller et al. [7]. The way we patch

```
res = dwarf_offdie(dbg, offset, &array_die, &error);
res = dwarf_child(array_die, &return_kid, &error);

res = dwarf_attrlist(return_kid, &atlist, &atcnt, &error);
for(i = 0; i < atcnt; i++){
    res = dwarf_whatattr(atlist[i], &return_attr, &error);
    if(res == DW_DLV_OK && return_attr == 47)
        res = dwarf_formudata(atlist[i], &form_data, &error);
}
```

Figure 6.18: Function used for getting the size of an array using the DPL tool. The first step is to get the DIE child of the variable declared as an array, then we get all of its attributes and if an attribute is of *type 47* (type for arrays in DWARF) then we look for the array size and return it.

source code is by following the **jmp** heuristics of x86 and *jumping* to a new function that represents the code we want to patch. In other words, we minimized the task of hot–patching source code by *pointing* the function that we wish to patch to a new address that contains the new piece of code.

Most of the security patches analyzed in Chapter 3 constrain themselves to modify a single function. One test case for hot–patching source code consisted of an application with a new function that includes the source code of the patch. By *pointing* or *jumping* the old function to the new function's address we are able to include the new source code in the running application.

6.4.4 Summary of DPL System

In Chapter 7, we simulate five different security patches using a test suite that consists of running a basic program and a patch for that program (representing what the security patch is supposed to update). In order to patch the source code, we use a technique that jumps to a new function on each test case. The data patch consists of an SQL script and a function address that patches the source code of the application by jumping to a new address. The entire workflow of the DPL tool is explained in Figure 6.13. We can also summarize this workflow as follows:

1. Attach to a running application by giving its Process ID (PID).

2. Create a MySQL database to store every data structure from the application.

3. Pass the data patch to DPL as a parameter and update the database according to it. The data patch consists of a set of SQL statements that result from the translation of the patch into a data patch.

4. Export all the data structures back to the application by updating their addresses.

5. Detach from the process and continue running it.

6.5 Applications for DPL

Developers can use DPL in many different ways. These other applications of DPL include, but are not limited to, the following set:

1. A framework for developers to test their patches and possible conflicts with data structures.

2. For modifying complex data structures on the run (e.g., convert a linked list to a binary tree) – This is part of DPL's future work.

3. For having an organized database with information of every active local, global and dynamically allocated data structure.

6.6 Limitations of DPL

Limitations for DPL are considered as sources of error, or sources of false negatives or false positives. We also enumerate some limitations of the DPL System The following are the set of limitations for the DPL model:

1. When DPL is not able to express patch semantics in our language, it might imply that:

    - The patch may contain certain elements that have an unpredictable impact on data semantics (e.g., input, or unbounded computation).
    - The patch may be complex and difficult to categorize or classify.
    - The patch may be broken.

2. For our machine learning experiments we had a limited dataset and the calibration of our SVM might not have been good enough to predict with a 100% of accuracy.

3. Manual error or human error in manual classification and in the assessment of patch characteristics.

4. The translation of C source code to SQL statements might be considered a limitation. For future work, we plan to deploy a module for automating this task.

5. DPL is limited to the power of SQL as a language for expressing data modifications.

6. We need to compile every application we want to analyze with DPL by using the *-g* flag on GCC to get the debugging symbols.

7. Users that want to run DPL on their system need to install Pin and our Pin tool.

Chapter 7

Evaluation Methodology

In this chapter we evaluate the analysis, design, and implementation of our proof of concept for the DPL system, as well as our empirical study and automation of classifying security patches. This chapter's approach is to evaluate our hypotheses, as well as to answer the research questions stated in Chapter 1. With the experiments presented in this chapter, we want to evaluate various aspects of this thesis independently and describe their relationship with other aspects of our investigation.

In this evaluation of our current system, we are interested in studying: what type of security patches are feasible to be patched using DPL's heuristics, how many types of data modification statements we can patch, what types of application we can patch using our proof of concept, and how DPL can patch larger and more complex applications.

7.1 Evaluating our Empirical Study

In Chapter 4 we analyzed our dataset of 75 security patches of different open–source applications. We considered this to be our empirical study and it gives an understanding of how security patches update data semantics of data structures by using different groups of statements. We classified these groups of statements according to how they modify data, constructing this way our set of data modification machines (DMMs). With this empirical study, we mean to evaluate two of our hypotheses. We were also able to expand our set of research questions by automating the process of classifying a patch as feasible or infeasible. For this automation we used a machine learning algorithm called support vector machine (SVM). The hypotheses we are revisiting in this section are:

Hypothesis A: *Security patches modify a small set of statements related to data structures therefore we can classify data structure modifications into different categories that we defined as a set of data operations..*

Hypothesis B: *We can classify security patches according to the feasibility of hot–patching their data semantics.*

7.1.1 Analysis of Patches

Our dataset consists of 75 patches that we classified into three categories: patches that are infeasible because we cannot express their statements (e.g., they expect arbitrary computation such as user input and library calls), patches that are not interesting for our purposes because do not modify data structures (i.e., they have a 0% of data modification statements), and patches that are considered to be feasible.

After manually analyzing and studying our dataset of 75 patches, we concluded that there are similar operations modifying data structures in 38 out of the 75 patches. Of the remaining patches, 24 are not modifying any data structures, and the rest are infeasible to be implemented by using our DPL system. After analyzing the patches, we confirmed our hypothesis that most of the patches have very small and constrained statements, agreeing also with Arnold et al. [4]. Some of these statements modify data structures using similar operations. We concluded that we were able to express these statements in a more general form by classifying them according to their heuristics for modifying data structures, this way demonstrating our Hypothesis A. This classification of data operations is expressed as our set of DMMs explained in Chapter 4, more specifically in Table 4.1.

For Hypothesis B, we decided to automate the classification of security patches, according to their feasibility for hot–patching data structures, by using machine learning algorithms. We decided to use SVMs as our machine learning algorithm, and thus we needed to represent our security patches in a feature–vector form. We translated the security patches into an intermediate graphical language to get a feature vector. We then applied PCA to map our 15–

dimensional feature vector into a 4–dimensional one, this way having a better understanding of what are the most crucial features of a security patch. According to PCA, the most independent features are:

- **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 4.1, the longest path would be of 5 from the starting node to the free() operator.

- **Number of 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 input degree a node has in the graph. The inputs are considered to be, for example, return values that modify a variable, or variables that are modified inside a loop or a branch. In Figure 4.1, the maximum value for an input node degree is 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 the value of this feature 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).

After getting a final 4–dimensional feature vector, we applied a SVM algorithm to classify patches according to the feasibility of implementing them using the DPL system.

7.1.2 Applying SVMs for Predicting Feasibility

For studying Hypothesis B, we analyzed how our SVM classifies patches according to these features, and if it classifies security patches correctly according to our definition of feasibility. For doing it, we experimented with our dataset in two different ways. First, we divided our dataset into a training dataset (2/3 of our data, or 50 patches) and a testing dataset (1/3 of our data, or 25 patches). 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.

On the second experiment, we randomly selected 5 subsets of our dataset (30 patches) and defined them as our training data, and the remaining 45 patches as our testing dataset. We had 5 different combinations of our dataset and predicted the labels for our testing dataset by using the same heuristics as before (that is, by using 5 different kernels). Our purpose was to divide our dataset in two different approaches, (1) learning from the majority of our dataset (2/3), and (2) training with the majority of our dataset (3/5). We finally did a leave–one–out approach as a cross–validation technique, the results for this experiment are presented in Figure 8.8.

We used a function in the R language to train and test our SVM [31]. After applying PCA over our data, we get our original space mapped to a 4-dimensional space. We decided to choose the features 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. As a reminder, the three categories chosen for our features were: control flow, data operations and graph representation.

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. This experiment helped us understand Hypothesis B, and its results are discussed in the next chapter.

7.2 Using DMMs for Security Patches

DMMs are a key part of our research. They are the result of studying over 75 security patches, and they provide a classification for statements that modify data structures. However, DMMs by themselves are only a description of a group of statements that modify data structures using a certain procedure (e.g., validating first an if case, or inside a while loop). They do not provide any description of how to patch data structures using the DPL system. For them to be implemented in our system, they first need to be translated into SQL statements.

For using our set of DMMs for dynamically updating data structures, we need to map our set of DMMs to the group of statements that is modifying data structures in our set of patches. We want to demonstrate that DPL is capable of patching data structures according to our translation of DMMs and security patches into SQL statements. For this, we developed a small application and attached to it using DPL to modify the data structures according to the patch.

We want to demonstrate that the data modification machines from Chapter 4 define a set of operations for expressing different patches as data patches. More precisely, that most of the statements from a patch, related to data modifications, can be expressed by using our set of DMMs. This bring us to our third hypothesis:

Hypothesis C: *We can use these data operations to dynamically update security patches according to how they modify data semantics.*

It is worth mentioning that some statements were not translatable to SQL, such as the values being returned from other libraries' functions, or macros that modify data structures. For the first set of statements, we can model these functions in our test suite by mimicking their respective functionality. We can also trace the value of a certain macro and express it inside our data patch object.

7.2.1 Patching our Data Modification Machines

Let us revisit Table 4.1. This table illustrates the definition of different DMMs. We can also express our DMMs as C source code and as a set of SQL statements. The purpose of this subsection is to illustrate how each DMM works and how to patch them by using DPL. By the end, we were able to patch these data modification machines by querying the database and following the heuristics defined in Chapter 6. The advantage of defining DMMs is that we can combine them in many different ways to express data modifications in security patches. The most basic machines are explained below:

Machine #1: Simple Data Modification

Simple data modification (SDM) statements appear in patches as new pieces of source code that change the value of an already existing variable. The fact that it changes an already existing variable is important, since it differentiates them from Machine #2. SDM is the most important machine of our set of machines, because it defines a data operation machine (i.e., all DMMs have a SDM in their set of operations that define them as a DMM).

Example in C	Example in SQL
variable = 0;	**UPDATE** *LOCAL_VARS* **SET** *Value = 0* **WHERE** *Name = "variable"*

Table 7.1: An example of statements that use DMM#1 in security patches expressed in C source code and how they get translated into SQL statements.

Machine #2: Declaration of a New Variable

Declaring a new variable is a statement commonly found in security patches; it can also be defined as the *creation* of a new variable by declaring its type, name and values – the metadata DPL's database uses. It differs from SDMs by substituting the **search** basic operation for a **create** operation.

Since these DMMs are creating new variables, and not modifying existing ones, we need to handle them in a different way that DMM#1. For this, we added a default address

Example in C	Example in SQL
int variable = 0;	**INSERT INTO** *LOCAL_VARS* **VALUES**("variable", "int", 4, "0xDEADBEEF", 0)

Table 7.2: An example of statements that use DMM#2 in security patches expressed in C source code and how they get translated into SQL statements.

(0xDEADBEEF) to the address field. When the DPL System gets a variable with this address it allocates memory space for it (according to the size of the variable) and then stores the respective value if necessary.

Machine #3: If case followed by data modification

This is one of the most common DMMs we encountered when analyzing security patches. It consists of an *if case* followed by a data statement such as a data declaration or data modification. We can express this statement by using a **search**, **compare** and **modify/create** statements. Two different cases are given in Table 7.3 and Table 7.4, the first one for data modification and the latter for data declaration.

Example in C	Example in SQL
if(case > 0) var = 10;	**IF** *Name = "case"* **AND** *Value = 0* **UPDATE** *LOCAL_VARS* **SET** *Value = 10* **WHERE** *Name = "var"*

Table 7.3: An example of statements that use DMM#3 in security patches expressed in C source code and how they get translated into SQL statements.

Machine #4: If/Else case followed by Data Modification

Similarly to Machine#3, this DMM includes an extra statement for comparing values if the case is not met. That is, if we get to the else case then we would have to use an additional **search** and **compare** operation. Two different cases are given in Table 7.5 and Table 7.6, the first one for data modification and the latter for data declaration.

Example in C	Example in SQL
if(case > 0) int var = 10;	**IF** *Name = "case"* **AND** *Value = 0* **UPDATE** *LOCAL_VARS* **VALUES**("var", "int", 4, "0x804939"', 10)

Table 7.4: An example of statements that use DMM#3 in security patches expressed in C source code and how they get translated into SQL statements. This time, instead of a simple data modification we declared a new variable.

Example in C	Example in SQL
if(case > 0) var = 10; else var = 20;	**IF** *Name = "case"* **AND** *Value = 0* **UPDATE** *LOCAL_VARS* **SET** *Value = 10* **WHERE** *Name = "var"* **ELSE** **UPDATE** *LOCAL_VARS* **SET** *Value = 20* **WHERE** *Name = "var"*

Table 7.5: An example of statements that use DMM#4 in security patches expressed in C source code and how they get translated into SQL statements.

Machine #5: Data Modifications using operators and functions

Many operators and functions from the C language can be expressed by using our metadata. As an example, we can express functions like free() by making the value of a particular address equal to zero, or the operator sizeof() by asking the database the size of a particular variable. These are also operators that can be encountered when analyzing patches.

Machine #6: Loops followed by Data Modification

The last DMM is focused on updating data structures that are being modified according to a loop condition. This was the only machine that we were not capable of updating with our primitive operators, and the standard query language. For updating this particular DMM, we had to expand our language to include the *loop* operator, and use PL/SQL[34] which is an extension to SQL. PL/SQL supports procedural programming and allows the use of loops inside MySQL.

Example in C	Example in SQL
if(case > 0) int var = 10; else int var = 20;	**IF** *Name = "case"* **AND** *Value = 0* **UPDATE** *LOCAL_VARS* **VALUES**("var", "int", 4, "0x8049394", 10) **ELSE** **UPDATE** *LOCAL_VARS* **VALUES**("var", "int", 4, "0x8049394", 20)

Table 7.6: An example of statements that use DMM#4 in security patches expressed in C source code and how they get translated into SQL statements. This time, instead of a simple data modification we declared a new variable.

Examples in C	Examples in SQL
var2 = sizeof(var1);	**SELECT** *Size* **FROM** 'LOCAL$_V ARS'$ **WHERE***Name =' var1'* $THEN$ **UPDATE***LOCAL_VARS* **SET***Value =' size'***WHERE** *Name =' var2'*

Table 7.7: An example of implementing the sizeof() operator with our System. After getting the size of the variable with the DWARF info, and storing it into the database, we can model this operator by asking for the size of a particular variable to the MySQL database.

Examples:

This demonstrates that our set of data modification machines can be expressed as set of SQL statements representing a data patch, and thus can be implemented by using the DPL system.

7.2.2 Using DMMs to Update Data Semantics According to Security Patches

After implementing the DMMs expressed in Section 4.4, we study how these DMMs can be implemented for patching data structures of our set of security patches. We chose five different patches for this purposes and created a test suite that emulates the source code that is being updated by the security patch in the application.

For our proof of concept, we developed a test suite for 5 security patches in Table 7.2.2

Example in C	Example in SQL
while(case) var = var + 1;	WHILE CONDITION DO: **UPDATE** *LOCAL_VARS* **SET** *Value = 'var's value +1'* **WHERE** *Name = 'var'*

Table 7.8: This Table illustrates how to implement a loop operation using our DPL System. In order to do this, we needed to extend the Standard Query Language to Procedural Query Language to include this kind of operations. We also extended our language to include a *loop* operator. This is the only DMM that has not been tested using our DPL implementation yet.

Name	Release Year	Application	Vulnerability
CAN-2004-0751	2004	Apache HTTPD	Denial of Service
CVE-2007-4138	2007	Samba	Grants root access
CAN-2004-0748	2004	Apache HTTPD	Denial of Service
CVE-2006-3403	2006	Samba	Denial of Service
AST-2012-005-10	2012	Asterisk	Heap Buffer Overflow

Table 7.9: Security Patches used to demonstrate the capabilities of DPL as a hot–patching system

and 5 applications that mimic the behaviour of the function being patched. We first decided to use a security patch for each application (3 security patches), but we thought it would be a small dataset for evaluating this experiment. Therefore, we then decided to choose 5 security patches, two from Apache HTTPD, two from Samba, and one from Asterisk. By selecting patches this way, we showcase how we are able to demonstrate that our set of DMMs work with different softwares. We also decided to study security patches fixing different vulnerabilities (denial of service, root access, and heap overflow) from different years (from 2004 to 2012).

We then express the security patch as a data patch object and applied them to the application by using the DPL system. The results for these experiments include the set of DMMs used for each patch, and a performance evaluation when updating the data structures using DPL. The results are discussed in the next chapter.

7.3 Patching Data Structures of Applications with DPL

There are many ways to implement data structures in the C programming language, and thus many ways to parse the debugging information. This is why patching applications, that are updated by the security patches presented in Table 7.2.2, is difficult to achieve using DPL. By now, DPL is a proof-of-concept that works well for demonstrating the heuristics we have defined throughout this thesis, and applying our concept of DMMs. In this section we demonstrate how DPL updates data structures of running applications in our test suite. We are also interested in evaluating the computational power of DPL in order to analyze how we can patch more complex applications by using the system.

Hypothesis D: *DPL has enough computational power to patch data semantics of a set of C applications.*

With this hypothesis we want to demonstrate that, by using our heuristics, our system is capable of dynamically updating an application by following these steps: getting the data structures that are currently live, or in–memory, in the application and storing them in the database, modifying the data structures according to the security patch, and exporting the modified data structures back to the application.

7.3.1 Patching C Applications

Our first experiment for modifying data structures was creating a simple *hello world* application with a global variable to be patched. The application was programmed to display two messages using two different functions. Each message displays the value of the global variable *variable* and, as illustrated in Figure 7.2, the variable is not being altered by the source code of the application.

We use a sleep of 10 seconds for attaching our DPL system before calling the subroutine bye(). Then, after attaching to the application, DPL uses its instrumentation routines for importing the data structures to the database. In Figure 7.5 we show the result from the

```
mysql> select * from GLOBAL_VAR;
+-----+----------+-------------+---------------------+----------+-----------+------------+
| ID  | name     | type_name   | type                | bytesize | address   | value      |
+-----+----------+-------------+---------------------+----------+-----------+------------+
| 187 | variable | int         | DW_TAG_base_type    | 4        | 134520856 | 8          |
| 230 | var2     | int         | DW_TAG_base_type    | 4        | 134520864 | 76         |
| 248 | var3     | unsigned int | DW_TAG_base_type    | 4        | 134520868 | 78         |
| 266 | var4     | long int    | DW_TAG_base_type    | 4        | 134520872 | 2832       |
| 309 | var6     | char        | DW_TAG_base_type    | 1        | 134520888 | 97         |
| 327 | var7     | char        | DW_TAG_pointer_type | 1        | 134520892 | 64         |
+-----+----------+-------------+---------------------+----------+-----------+------------+
8 rows in set (0.00 sec)
```

Figure 7.1: Some of the types of variables that we are able to recover and patch. For the characters, we get their ASCII value and for the character pointers (var7) we only get the first character.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int variable = 8;

void hello(){
    printf("Hello World, the variable_ is %d\n", variable);
}

void bye(){
    printf("See you later World! Your variable is %d\n", variable);
}

int main(){
    hello();
    sleep(10);
    bye();
}
```

Figure 7.2: Source code used for patching a hello world program using the DPL system.

import routine and how it effectively stores the global variable we want to patch into its respective table. Then, we use the data patch object illustrated in Figure 7.4 for *patching* the global variable.

The result is shown in Figure 7.3, the first function runs the print message with the variable's original value and the second function the new value. After finishing the sleep command, the second function gets called printing out the variable's new value. DPL is capable of importing, patching, and exporting every data type from the C language.

7.3.2 Patching More Complex Data Structures

DPL is capable of hot–patching more complex data structures than the primitive data types of the C programming language. We decided to use the linked list implementation, illustrated in Figure 7.7, for this experiment. The linked list consists of two integer elements, rc and p,

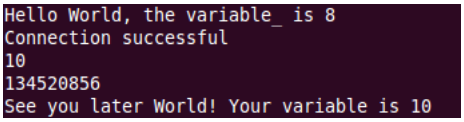


Figure 7.3: This Figure shows the output of the hello world program. The first statement prints out the variable **before** hot–patching it. The second statement states that our DPL System is attached to the process. The third statement shows the new value we want to introduce followed by the address of the variable. Finally, we output a message with the new value of the same variable. The source code for this program is illustrated in Figure 7.2, and its data patch object in Figure 7.4

```
UPDATE GLOBAL_VAR SET value = 10 WHERE name = "variable" AND value = 8;
SELECT value, address FROM GLOBAL_VAR WHERE name = "variable";

VARIABLES:
variable

TABLES:
GLOBAL_VAR
```

Figure 7.4: Data patch object example for the hello world program in Figure 7.2.

```
mysql> select * from GLOBAL_VAR;
+-----+----------+-----------+------------------+----------+-----------+-------+
| ID  | name     | type_name | type             | bytesize | address   | value |
+-----+----------+-----------+------------------+----------+-----------+-------+
| 137 | variable | int       | DW_TAG_base_type | 4        | 134520856 | 8     |
+-----+----------+-----------+------------------+----------+-----------+-------+
1 row in set (0.00 sec)
```

Figure 7.5: Example of table in our database after importing a global data structure.

```
mysql> select * from PTR_STRUCT;
+---------+-----------+--------+-----------+----------+
| name    | data      | offset | address   | type_tag |
+---------+-----------+--------+-----------+----------+
| rc      | 9         | 778    | 146358580 | 36       |
| p       | 12        | 790    | 146358584 | 36       |
| rc      | 8         | 778    | 146358540 | 36       |
| p       | 12        | 790    | 146358544 | 36       |
| rc      | 7         | 778    | 146358500 | 36       |
| p       | 12        | 790    | 146358504 | 36       |
| rc      | 6         | 778    | 146358460 | 36       |
| p       | 12        | 790    | 146358464 | 36       |
| rc      | 5         | 778    | 146358420 | 36       |
| p       | 12        | 790    | 146358424 | 36       |
| rc      | 4         | 778    | 146358380 | 36       |
| p       | 12        | 790    | 146358384 | 36       |
| rc      | 3         | 778    | 146358340 | 36       |
| p       | 12        | 790    | 146358344 | 36       |
| rc      | 2         | 778    | 146358300 | 36       |
| p       | 12        | 790    | 146358304 | 36       |
| next    | 0         | 802    | 146358308 | 15       |
| st_head | 0         | 816    | 146358312 | 15       |
| next    | 146358296 | 802    | 146358348 | 15       |
| st_head | 0         | 816    | 146358352 | 15       |
| next    | 146358336 | 802    | 146358388 | 15       |
| st_head | 0         | 816    | 146358392 | 15       |
| next    | 146358376 | 802    | 146358428 | 15       |
| st_head | 0         | 816    | 146358432 | 15       |
| next    | 146358416 | 802    | 146358468 | 15       |
| st_head | 0         | 816    | 146358472 | 15       |
| next    | 146358456 | 802    | 146358508 | 15       |
| st_head | 0         | 816    | 146358512 | 15       |
| next    | 146358496 | 802    | 146358548 | 15       |
| st_head | 0         | 816    | 146358552 | 15       |
| next    | 146358536 | 802    | 146358588 | 15       |
| st_head | 0         | 816    | 146358592 | 15       |
+---------+-----------+--------+-----------+----------+
```

Figure 7.6: This figure illustrates the results of importing the linked list implementation in Figure 7.7. DPL was able to recover every field of the linked list declared in the application, as well as their respective values and addresses.

and one struct pointer, *next*. We dynamically allocated the linked list and then recovered all of its fields and stored them in a table called PTR_STRUCT, illustrated in Figure 7.6.

As Figure 7.6 shows, we were able to recover the two integer members of the linked list and their respective values, and addresses. We are also able to modify any value of a particular member, and export the value back to the application by referencing the member's address field.

Recovering and modifying dynamically allocated data structures is one of the most challenging tasks DPL is capable of achieving. By working with the values and addresses of the variables, we are able to make a particular node equal to NULL, or modify the value of a particular node.

```c
struct LINKED{
    int rc, p;
    struct LINKED * next;
};

int subs(int mts, int cms){
    int i;
    struct LINKED instance1;
    struct LINKED * inctx = NULL, * st_head = NULL;

    for(i = 1; i < 10; i++){
        inctx = (struct LINKED *) malloc(sizeof(struct LINKED));
        inctx -> rc = i;
        inctx -> p = 12;
        inctx -> next = st_head;
        st_head = inctx;
    }

    printf("%d -> %u\n\n", inctx -> rc, &inctx -> rc);
    int length = return_add(3, 4);

    return inctx -> rc;
}
```

Figure 7.7: An example of a linked list implementation that was correctly patched by the DPL System.

7.3.3 Analysis for Patching More Complex Applications

DPL, unfortunately, has some limitations. One of DPL's limitations is the lack of compatibility with multi-threaded applications, and the complexity of getting data structures for *large-pieces* of source code that interact with other C libraries. DPL has some technical limitations which include variables being returned from library calls, working with macros from other libraries, and character pointers. As mentioned before, we also need the debugging symbols for every program we are patching using DPL.

However, the heuristics that DPL would use with a larger application remain the same. Therefore, applying a patch to Firefox, Asterisk, Samba, or an even bigger software (e.g., operating systems, web servers, or other frameworks) consists of the same steps we follow for our Test Suite. The difference would remain in the mechanism we use to import the data structures to the database, and a bigger complexity when getting the local, and dynamic allocated variables. Figure 7.8 illustrates the process for updating any application, including the applications we are not able to patch because of technical limitations. If we want to be able to patch larger, and more complex applications, we would need to improve the second step of the diagram in Figure 7.8.

For future work, we want to study how to implement DPL in these applications. We are

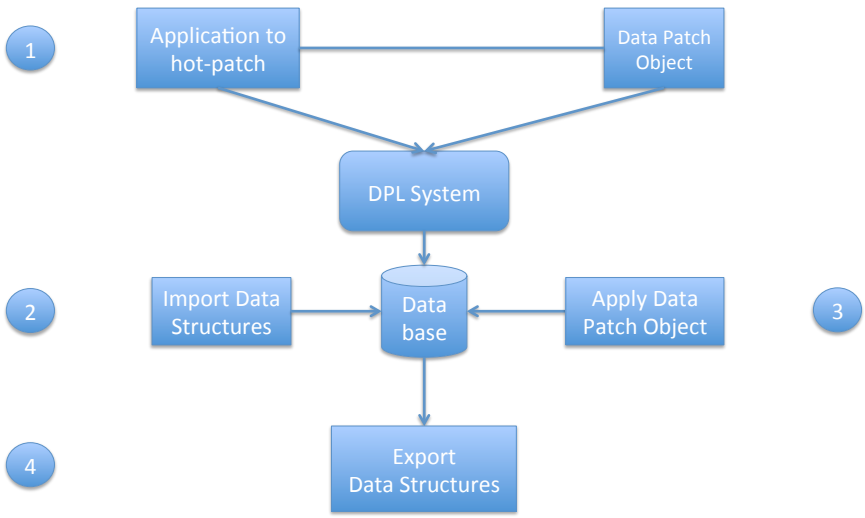


Figure 7.8: This figure illustrates the process of hot–patching any application by following 4 different steps. The first step (1) is to give the application to be patched, and a translation of a security patch to a data patch object as input parameters to DPL. After this (2), DPL imports every data structure into the database that is in communication to DPL, this is the most challenging routine of DPL and the one that is limiting our set of experiments. We are planning to improve this routine and be able to import data structures from any application, including the ones from our security patch examples. Then (3), the data patch object is applied to the database as MySQL statements, modifying this way the data structures that were previously imported. Finally (4), the data structures are exported back to the application, and thus patched according to the security patch.

studying implementing DPL on mobile applications in the near future. We think that this field is constantly becoming more popular, and one of its main disadvantages is the need of, not only restarting, but completely paralyzing an application while it updating it.

7.4 Research Questions

After analyzing our set of security patches in Chapter 4, applying a SVM algorithm in Chapter 5, and evaluating our set of experiments in this chapter we expanded our initial set of research questions to include the following:

1. How well does the machine learn to predict results according to the heuristics we use? By getting the resulted set of labels we match them with the manual labels we used for our analysis of the dataset.

2. What defines a patch to be feasible according to our SVM? Finally, we reply to this question by analyzing what the SVM defined as feasible and what the patches labeled as feasible or infeasible have in common.

3. Can we express our set of data operations in our current system? We answered this question by translating our set of DMMs into SQL statements in Section 7.2.1.

4. Can we use our current system for other purposes than hot patching? We can also use DPL for different purposes, such as modification of complex data structures such as the linked list presented in Section 7.3.2. For future work, we can deploy DPL as a test suite for developers to test patches before deploying them.

Chapter 8

Results

This thesis consists of three different main components that, by interacting with each other, ultimately make DPL capable of patching data structures. These components are the analysis of patches, applying a machine learning algorithm and the implementation of the framework.

In this chapter we present the results for each component and how these results help us to demonstrate the value of the DPL system. At the end we discuss how we could make hot–patching a feasible activity according to our results and analysis.

8.1 Analysis of Patches

The results for this work are presented as a comparison between the manual classification of patches and the SVM results. Figure 8.1 compares the manually classified testing data with our SVM results.

After manually analyzing seventy-five different patches, we concluded that the C language modifies data structures using the same set of operators, and that most of these data modifications can be expressed by using simple operators such as *read, write, compare* and *search*. Our proof of concept consists of a framework that connects to a SQL database and, by using standard SQL queries, is able to express most of these data modification statements.

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 8.1: Results for the classification of our training data set with no labels.

Longest Path	max in degrees	cycles	Percentage	Label	Polynomial	Radial Basis	Linear	Laplacian	Anova
Infeasible patches: Categorized as infeasible by the SVM and our manual analysis									
-0.75	-0.5	-1	-1	N	N	N	N	N	N
-0.5	-0.5	-1	-1	N	N	N	N	N	N
-0.25	-0.5	-1	-1	N	N	N	N	N	N
-0.75	-0.5	-1	-1	N	N	N	N	N	N
-0.25	-0.5	-1	-1	N	N	N	N	N	N
-0.5	-0.5	-1	-1	N	N	N	N	N	N
-0.75	-0.5	-1	-1	N	N	N	N	N	N
-0.5	-0.5	-1	-1	N	N	N	N	N	N
-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									
0	0	-0.4	1	N	Y	N	N	N	N
-0.75	0	-1	1	N	Y	Y	Y	Y	Y
0.25	0	-0.2	1	N	Y	N	N	N	N
-0.5	-0.5	-1	1	N	Y	Y	Y	Y	Y
Feasible patches: Categorized as feasible by the SVM and our manual analysis									
-0.5	0	-1	0.6	Y	Y	Y	Y	Y	Y
-0.5	0	-1	0.5714	Y	Y	Y	Y	Y	Y
-0.5	0	-1	0	Y	N	Y	Y	Y	Y
-0.5	0	-1	0.5	Y	Y	Y	Y	Y	Y
0	0	-1	1	Y	Y	Y	Y	Y	Y
-0.75	-0.5	-1	1	Y	Y	Y	Y	Y	Y
-0.75	-0.5	-1	1	Y	Y	Y	Y	Y	Y
-0.25	0	-1	1	Y	Y	Y	Y	Y	Y
-0.5	0	-1	0.6	Y	Y	Y	Y	Y	Y
-0.25	0	-1	0.6	Y	Y	Y	Y	Y	Y
0	0.5	-1	1	Y	Y	Y	Y	Y	Y
0	0.5	-1	1	Y	Y	Y	Y	Y	Y
				% of Match:	80%	92%	92%	92%	92%

Figure 8.1: Comparison between manual 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.

The last two questions can be answered with our SVM results. Table 8.1 and Table 8.2 contain our results. According to this data, SVMs is a good technique 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% 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% correctness. We ended our set of experiments with SVM by taking a leave–one–out

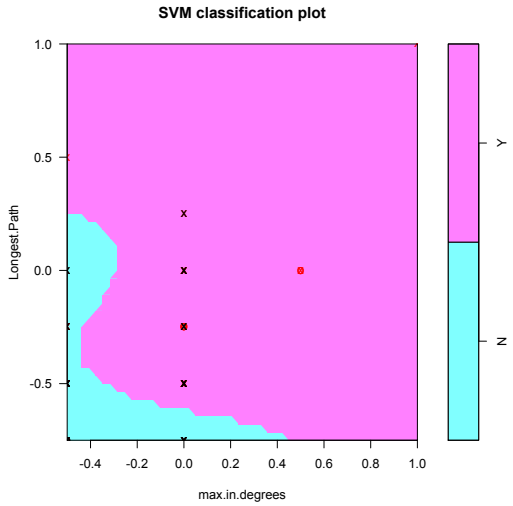


Figure 8.2: SVM classification to three different comparisons of two attributes using a radial basis kernel function. The X points are the support vectors.

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 8.2: Results for Testing Data Set using our first experiment (1/3 of dataset).

approach as a cross–validation technique. A subset (50 patches) of this approach is presented in Figure 8.8. The results for the prediction using a leave–one–out experiment are of 82% correctness. These results are similar with the percentage of correctness of our first two experiments.

The results on Tables 8.2 and 8.1, tells us the best kernel functions to use with our dataset are radial basis and Laplacian. We have to 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.

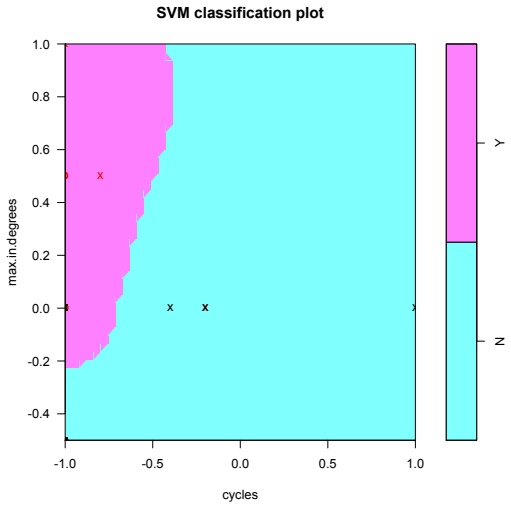


Figure 8.3: SVM classification to three different comparisons of two attributes using a radial basis kernel function. The X points are the support vectors.

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 8.3: Results for testing data set using our second experiment (3/5 of dataset).

8.1.1 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 8.1 and see that there is a notable relationship between the feature "percentage of data statements" and

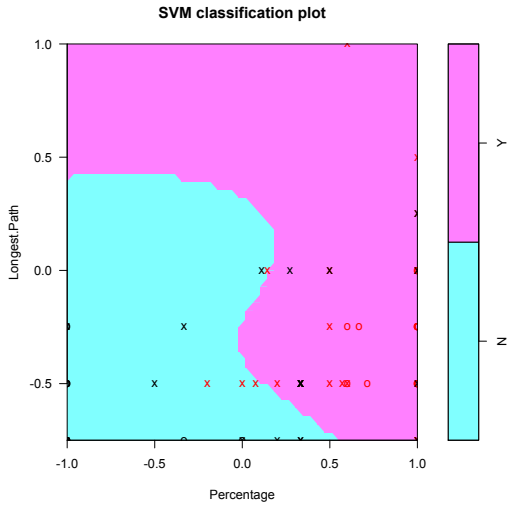


Figure 8.4: SVM classification to three different comparisons of two attributes using a radial basis kernel function. The X points are the support vectors.

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, and thus their data structures did not need to be updated. That leaves us with 12 **feasible** patches which all had over 50% data operation statements. Most of the 12 feasible patches had a bigger value for the "maximum input degree" (i.e., values returned from functions and stored in a variable) feature than the infeasible patches; this means that the more complex the control flow is, the harder it is to hot patch data structures. Even though we are able to express loops by expanding the computational power of our language, there are also no cyclic operations (i.e., for or while loops) in what the SVM classified as feasible patches. Another feature that is related to infeasibility is the longest path of a statement (i.e., illustrated in Figure 4.1 in Chapter 4) which is also related to the control flow of a patch. This feature tells us that the deepest the scope of an operation becomes (e.g., an if case inside an if case) also introduces control flow changes that may result in conflicts.

In conclusion, control flow changes increases the complexity of an application, and thus increases the possibility of getting conflicts in data structures after hot–patching.

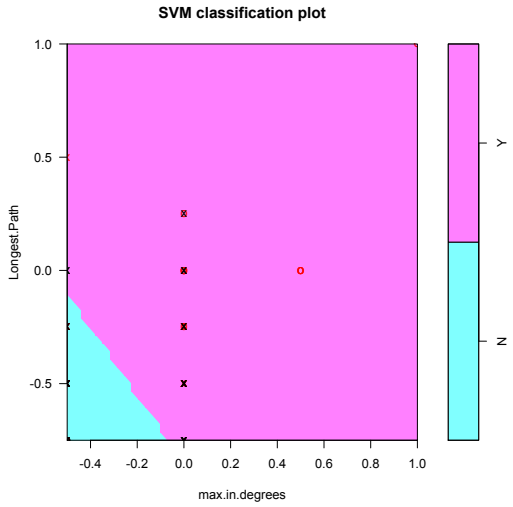


Figure 8.5: SVM Classification to three different comparisons of two attributes using a linear kernel function. The X points are the support vectors.

Answering the research questions that we posed above, we can conclude that we are able to express every data modification in 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.

The attributes we decided to use for each graph were helpful for our SVM to decide feasibility. There was no relationship between our set of features and the feasibility of a patch, which tells us that we could deduce feasibility of a patch, according to data modifications, by using our heuristics.

8.2 Discussion of the DPL System

One of the main limitations of DPL is understanding which patches are feasible to be implemented, this is why we decided to use a machine learning approach to predict the feasibility of a security patch. More than *predicting the feasibility*, we were analyzing which features create more conflicts when hot–patching. We got to the conclusion that when patch developers call functions that return variables they will be creating a more complex control flow

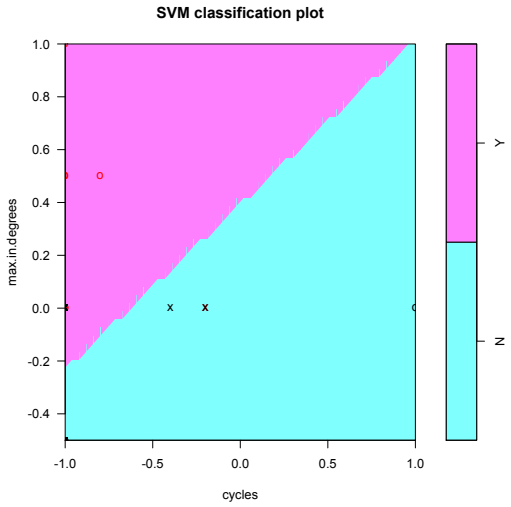


Figure 8.6: SVM Classification to three different comparisons of two attributes using a linear kernel function. The X points are the support vectors.

on the patch, hence making it harder to achieve hot–patching.

Our DPL system is a proof of concept that is not capable of hot–patching the applications from the security patches we studied. However, we created a test suite that was presented in Chapter 7 In this section, we study different experiments we run on the DPL system.

8.2.1 Revisiting our data modification machines (DMMs)

In Section 4.4, we presented our concept of **data modification machines** which can be described as a collection of statements, of a security patch, that modify a data structure of the application. In Table 4.1, we present 7 different collection of statements that modify data that were commonly found in security patches. We then defined this statements as our set of DMMs. In the following table, we present the results for patching these DMMs using DPL. We added a new DMM (DMM#6) for working with loops that are somehow modifying a data structure. This is the only DMM that we were not capable of patching by using the standard query language from MySQL. We needed to expand our language for dealing with this particular machine by including a *loop* operator. We used an application with 138 lines of source code which implemented several functions, a linked list, global variables, dynamic

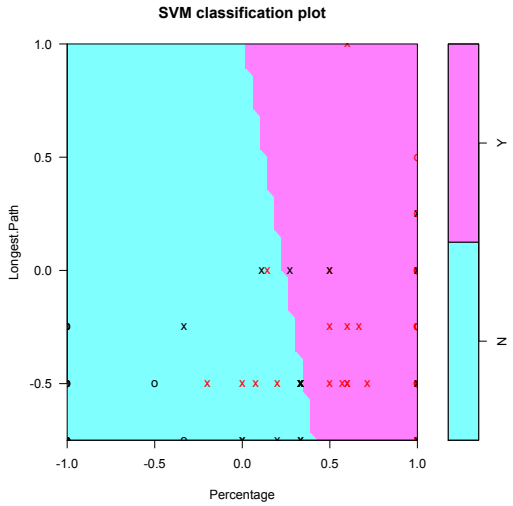


Figure 8.7: SVM Classification to three different comparisons of two attributes using a linear kernel function. The X points are the support vectors.

variables, and local variables for testing our DMMs.

For DMM#2, we had to find our own heuristics for declaring new variables inside the running application. For this, we added an address value of 0xDEADBEEF to the address field of the variable we want to declare. For example, if we want to declare a new variable of type *int*, with a value of 0, and name *var* in the running application, we issued a INSERT INTO Table(name = "var", type = "int", value = 0, address = "0xDEADBEEF"). This way, once we export the data structures back to the application, whenever we find an address

Machine	Name	Can we patch it?
DMM#1	Simple Data Modification	Yes
DMM#1	Simple Data Modification (2)	Yes
DMM#2	Declaration of New Variable	Yes
DMM#2	Declaration of New Variable (2)	Yes
DMM#3	If case + Data modification	Yes
DMM#4	If case + Else case + Data modification	Yes
DMM#5	Data Modifications using Operators	Yes
DMM#6	Loops + Data Modification	By expanding to PL/SQL

Table 8.4: This Table represents the different DMMs that we are able to use for hot–patching data structures with DPL.

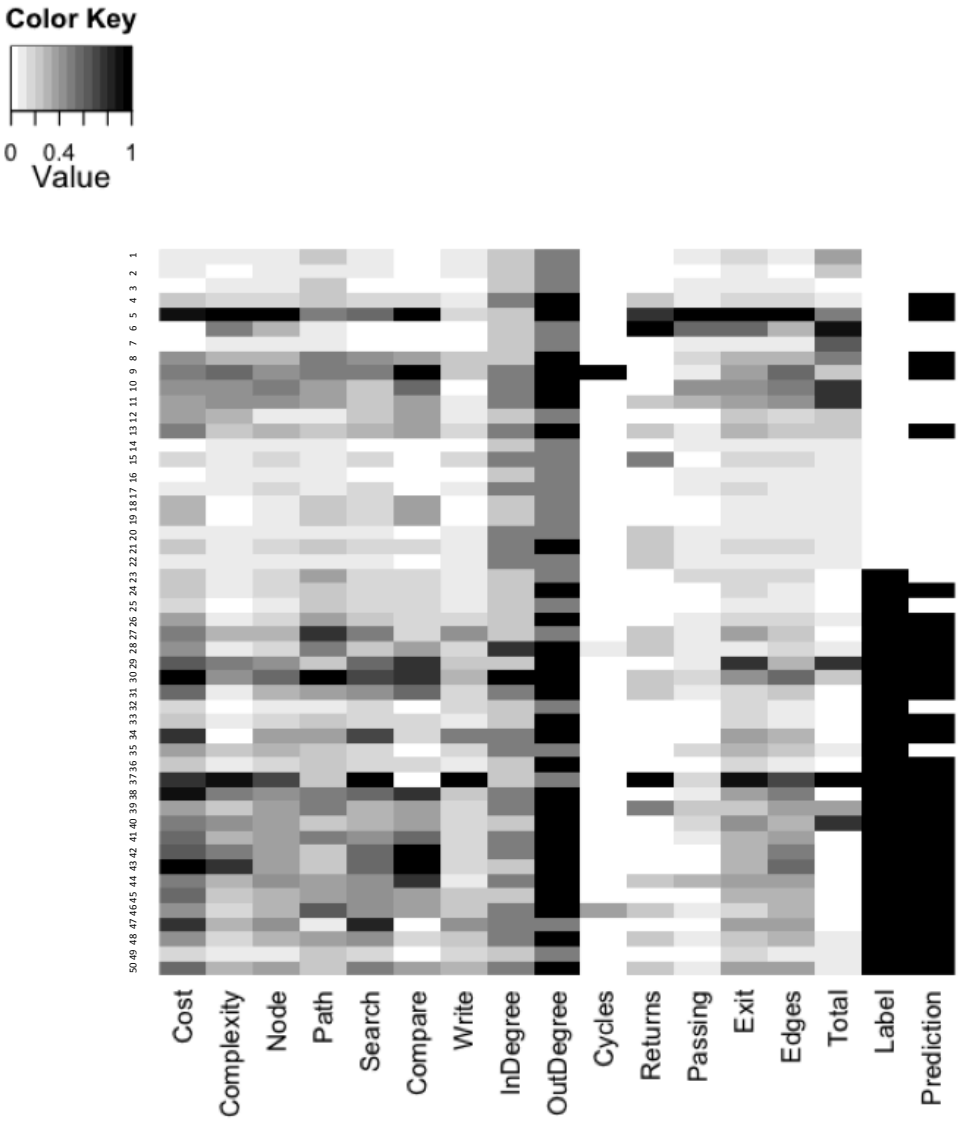


Figure 8.8: This figure shows a heat map representation of 50 feature vectors of security patches in our dataset. The darkness of each tile represents the complexity of the feature (i.e., how large is the feature), darker tiles represent larger features. The last two columns of the figure represent our manual classification (left) vs the SVM classification (right). The correctness of the results was consistent with the correctness of our previous experiments. Some patches (e.g., patch#5 in the figure) have complex features and confuse our SVM, other patches (e.g., patch#10) have large number of cyclic operations that also confuse our SVM.

of 0xDEADBEEF, we allocate memory in the program for that variable.

The limitation we had when trying to implement DMM#6 is that simple SQL does not support loop conditions in their standard language. For this, we had to use an extension called PL/SQL [34] that supports loops. This extends our language set of operators to: {*search, compare, read, write, loop*}. In Table 8.4, we present a description of the DMMs we are able to apply dynamically using our system.

8.2.2 Patching real-life security patch models

Our implementation of DPL is a proof of concept on where we implemented what we considered to be the four routines for hot–patching data semantics. These routines are illustrated in Figure 7.8. The purpose of the system designed in this thesis was to implement these routines and study different cases that can occur when hot–patching the applications of the security patches we analyzed. We concluded that, because we focused on hot–patching data structures, we can define these cases as the different data modification operations we found when analyzing how security patches modify data semantics. We found that we could define the commonly–used statements for modifying data structures as a set of artefacts we called DMMs. This set of DMMs is what we consider to be *feasible statements*. The purpose of our set of DMMs is to avoid statements that introduce radical control flow changes such as the ones found to be *infeasible* by our SVM.

Since DPL still has some technical limitations that makes hot–patching large software infeasible, we decided to model the statements the security patch is patching inside one of our test applications (Application #6 in Table 8.6). For doing this, we applied one by one the data patch representation of the security patches to our application. The data patch consisted of the DMMs SQL statements we need to use in order to patch the data structures according to the security patch. The results for this, including a performance overhead, and the DMMs that were used to achieve the task, are presented in Table 8.5. Since we are mostly focusing on hot–patching data structures, and not source code, the heuristics we

Name	Machines used	New Statements	Patched?	Performance
CAN-2004-0751	DMM#4	6	Yes	0.05 sec
CVE-2007-4138	DMM#1	1	Yes	0.03 sec
CAN-2004-0748	DMM#3	4	Yes	0.05 sec
CVE-2006-3403	DMM#3	5	Yes	0.05 sec
AST-2012-005-10	DMM#2 + DMM#4+ DMM#5	9	Yes	0.07 sec

Table 8.5: Results for applying our set of DMMs to Application#6 in Table 8.6. The performance is measured as the time it takes to import, hot–patch, and export a variable according to the DMMs. The performance is the average CPU execution time after running the DPL system 5 times with the respective data patch. The CPU model we use is an Intel(R) Core(TM) i7 CPU running at 2.00GHz in a 32–bit architecture, a cache size of 6144 Kb, and Ubuntu 2.6.32 as an operating system.

think are most important are the modification of the data structures in the form of a data patch.

This experiment depends on the previous experiment presented in Section 8.2.1 and shows that our DMMs are capable of hot–patching data structures according to a security patch, and that our procedure for hot–patching can interact with source code representing an application.

We first re–visited the security patches and defined the set of DMMs they needed to use for hot–patching the application's data structures. The performance overhead for patching these security patches is very low, taking between 0.03 and 0.07 seconds. We applied these patches to our main test case (Application #6 in Table 8.6) which consisted of 140 lines of code and 8 different functions with several local variables declared in their scope. We tested how DPL modifies the statements of each DMM by combining them in different ways to update data structures of our application.

8.2.3 Modifying data structures with DPL

Another of our experiments is trying to update several data structures from different programs. The purpose of this experiment is to understand how more complex data structures

could be hot–patched in more complex applications than the ones in Section 8.2.2.

We learned that patching dynamically–allocated variables is a hard task because, in the example of our linked list implementation, it involves modifying a node from a large data structure. Developers should remember, at all times, that many complex data structures (e.g., trees, tables, lists) have different nodes represented as different data structures. When a security patch need to modify a data structure, they need to take care of every node representing that structure. For example, referring back to our example illustrated in Chapter 1, if the developers introduce a condition to modify the data structure (i.e., if case is met, then modify data structure) they need to handle all the nodes already declared in that data structure.

We argue that developers of security patches modify data structures forgetting that some data structures are a **collection of nodes**. Therefore, if we would have to modify a data structure, such as a linked list at runtime, our system needs to be capable of recovering all the nodes of that data structure, modify them according to the statement, and export them back to the application. With DPL, we are capable of doing that by using the metadata in the database tables as a reference.

8.2.4 Performance Evaluation

We tested DPL over a test suite designed using six different applications that implemented several data structures in different ways. The purpose of this test suite was to measure the time it takes to hot–patch a data structure of one of these applications. A description of each application is given in Table 8.6 along with the description of how we hot–patched them using DPL. We also tested our DMMs by applying their respective SQL statements over our Application #6. The results for this experiment are presented in Section 8.2.1.

The performance of the DPL system can be measured by how long it takes to import every data structure, hot–patch them using a DMM, and export them back to the application. The performance results were between the values of 0.01 and 0.07 seconds but they depend on

the complexity of the application we are patching. The more data structures, variables, subroutines, and lines of code that are defined, the longer it takes for DPL to finish its procedure. This is because we have to import every data structure from the application, making it a complex activity that depends on the complexity of the application more than the DMMs that we are using to hot–patch its data structures.

8.3 Wrapping it up: Answering our Research Questions

At the beginning of this thesis, we stated a set of research questions that got expanded once we started experimenting with our framework. The complete set of research questions is answered in this section.

1. **Can we summarize our empirical study of 75 security patches by selecting a set of common data operations?** The answer to this research question is **yes**. We are able to model patches as *data patches* by using our DMMs, which we defined as our set of common data operations.

2. **How many patches can we express as data patches?** We are able to express 38 out of 75 patches as data patches. From the remaining 37 patches, twenty–four patches are uninteresting to us because they do not modify the semantics of any data structure. The remaining patches (13) were not feasible to be expressed as data patches because of their complexity. This includes calling functions from libraries, working with macros, and complex control flow.

3. **Are we able to automate the task of selecting patches that are translatable to our set of data operations?** Yes, we can automate this task. We gave as input the training and testing datasets to a SVM algorithm and it was able to learn about the feature vectors and predict feasibility.

Application	#Functions	#Data Structures	Data Structures Hot–patched	Performance
#1	2	Global: 6 Local: 6 Pointers: 2	We imported every data structure successfully and hot–patched a global variable. Every variable was of a primitive type (e.g., int, float, char).	0.01 sec
#2	1	Global: 10 Local: 0 Pointers: 0	We modified the values of the global variables inside the function scope. We attached with DPL and successfully imported every variable and then hot–patched one of them.	0.01 sec
#3	1	Global: 0 Local: 2 Pointers: 0	We imported the two local variables and modified one of them using DPL.	0.01 sec
#4	1	Global: 2 Local: 0 Pointers: 0	We declared two global arrays with values inside and correctly imported them to our database using DPL. We then successfully hot–patched one of the arrays.	0.01 sec
#5	3	Global: 1 Local: 1 Pointers: 0	We declared a global structure and a local structure with two fields. We correctly hot–patched both of them.	0.03 sec
#6	8	Global: 6 Local: 21 Pointers: 1	This was the most complex application of our test suite with around 140 lines of source code. We correctly imported every data structure declared inside the program and hot–patched a linked list implementation.	0.05 sec

Table 8.6: Description of the different applications we used for evaluating the performance of DPL as a hot–patching system. The system specifications are the same as in Table 8.5

4. **How well does the machine learn to predict results according to the heuristics we use?** Our SVM matched our labelling results with over 80% correctness for our testing dataset and over 84% for our training dataset, which was treated as a new dataset by giving it to the SVM with no labels.

5. **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. One of our assumptions, expressed in Chapter 4, is that the computation of these functions will return successfully, but in reality it is hard to predict the behaviour of calling a function. 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.

6. **Can we model $real - life$ security patches as data patch objects to use with DPL?**

   Since DPL is not ready to be deployed as a system to patch a real–life software yet, we needed to study its performance by modelling security patches used for our empirical study. DPL was capable of patching the security patches by focusing only on what it is patching and not other control flow or data flow that could exist in the application. In other words, we were able to correctly patch the data semantics of a security patch by focusing on the semantics of the function being updated by the patch.

7. **What type of data modification statements are we able to express**

as a data patch consisted of SQL statements?

SQL (Standard Query Language) is a relational-algebraic language. We are not able to express every DMM in Table 8.2.1 without expanding our set of primitive operations (i.e., search, write, read, and compare) to allow loops. However, if we use the procedural language that SQL offers[34], we would be able to express every DMM, including loops and cyclic operations. This expands our set of operators to: {search, write, read, compare, and loop}. The *loop* operator takes as an input the condition for the loop case, and then does the remaining operations according to the patch (e.g., loop(condition) then search(variable) then modify(variable)).

8. **How many of these statements (DMMs) are we able to patch when using our heuristics?**

   We are able to patch every DMM defined in Table 4.1 when using the standard query language offered by MySQL, except for DMM#7. We were able to patch this DMM, however, by expanding the SQL language to include procedural statements. When we expand the language, we also expand our set of operations by adding a new operator called *loop*.

9. **Can we add new variables or new elements to a data structure?**

   We can use MySQL relational–algebraic approach to add new variables to the program. For this, we added an address value of 0xDEADBEEF to the variable entries on the database that we know we need to allocate in the program. For example, if we want to declare a new integer variable in the program, we looked for its respective entry on the database and added the value 0xDEADBEEF on its address field. Then, when parsing the variables inside the database for the export routine, if a variable has an address of 0xDEADBEEF we allocated

memory space using our system to create the variable.

10. **What is the performance and overall evaluation of our current system?**

    Our system shows a very good performance evaluation. The average time for executing a DMM to patch a data structure was between 0.01 and 0.05 seconds depending on the application we are patching. A detailed description of our performance evaluation is described in Section 8.2.4.

11. **Can we use our current system for other purposes than hot patching?**

    By now DPL is very constrained because of technical limitations, we are only able to hot–patch applications of our test suite. We have not learned how to deal with more complex statements such as library calls, arbitrary computation (e.g., user input), and macros. However, we plan to deploy DPL not only as a hot–patching framework but a test suite that lets developers hot–patch data structures of an application by following a security patch. This will allow application developers to study if a security patch might bring unwanted conflicts within the application's data structures.

8.4 Discussion

After studying the different experiments we explained in the previous section, we got to different observations that could aid the hot–patching research field. Our main observation is that, when finding a vulnerability, most patch developers address the vulnerability by patching the respective source code statements that introduce it. This, however, is not the best approach if we want to create a security patch *feasible* to hot–patch. We, as a community, need a better communication between patch developers and application developers in order to test if the patch is **feasible to hot–patch** before deploying them. If patch developers

are aware of the type of statements that introduce conflicts when hot–patching, we could make hot–patching a feasible activity.

1. Patch developers should study better the statements that they introduced into an application. Specially if those statements modify the control flow of the application. We found that, by using our heuristics, variables that get assigned a value returned by calling a function are the hardest statements to patch. It was one of the features that remained after applying PCA (maximum node input–degree), and the most important feature to classify a patch as feasible or infeasible. Many authors [25, 42, 3, 4] agree that there are patches that are infeasible to implement in their frameworks. We think this is because patch developers do not take into consideration how the statements they introduce change the semantics of an application. We hope that our research works as a fundament for patch development and helps them to avoid selecting statements that make the task of hot–patching infeasible.

2. Another statement we found hard to hot–patch was the declaration of new variables. That is, introducing variables that were not declared **before** hot–patching the applications. The way we handled this was by taking advantage of the relational–algebraic approach of SQL and our metadata obtained by the DWARF info. These are elements that are not presented in other hot–patching frameworks.

3. Finally, loops also change the control flow of a program. We are able to implement loops in our relational–algebraic approach as long as we extend our SQL language to include procedural queries. This increases the power of SQL as a language for expressing data modifications.

8.5 Future Work

The DPL system implementation is still a work in progress. We were able to demonstrate its functionalities by hot–patching data structures of basic C applications. However, we want to expand the set of applications we studied in this thesis by introducing more complex applications such as the ones of the security patches studied in Chapter 4. The difficulty of working with more complex applications is that, when we import the data structures to the database, we need to parse and analyze the current state of the application. This is a very complex task and its difficulty increases according to the complexity of the application. We also want to increase our dataset to include more security patches for our patch analysis process, and automate the translation of patches into feature vectors.

Conclusions

Patches issued to fix vulnerabilities in an application not only change the source code of the program but also the state of many of its components (e.g., data structures, control flow). Dynamically patching applications is a technique that has not been mainstreamed yet due to the complexity of updating the state of the application after being patched.

After analyzing a set of patches for different applications, we can conclude that patches modify data in a similar fashion (i.e., using similar operations to modify different structures) and that, if we can model these modifications by using a gadget or subprogram to illustrate them, we can address many different patches by using a pre-defined set of subprograms. We called these subprograms Data Modification Machines (DMMs) and their task is to update data structures according to the control flow the patch implies for that particular operation.

We then applied machine learning techniques over the set of patches that were studied and concluded that we can predict if a patch is feasible to be implemented, according to how it modifies data structures, by having it mapped to a feature vector. Our SVM predicted with over 75% success if a patch is feasible to be implemented using a hot patching framework – a patch is not feasible to be implemented if, after it has been hot patched, we get an inconsistent state in the application (e.g., out-of-date data structures). We think that the results show that our approach for translating patches into feature vectors is valid and that our machine can predict with over 75% if a patch is feasible or infeasible. We plan to improve this ratio by introducing new features to our vector, and new security patches to our dataset.

Finally, we designed and implemented a proof of concept for a *data structure hot patching* framework that communicates with a database and, by giving the application and a data patch object with a set of SQL statements as an input, dynamically patches a small application's data state and source code. We concluded that we are able to represent every DMM, except for the DMM related to loop operations which would need us to extend the SQL

language with a SQL-Procedural language syntax, and patch data structures according to how the patch modifies them. This is because operations like loops introduce more complex control flow to the application, making us expand our language to include these operations.

For future work, we would need to expand the SQL language with a procedural language in order to be able to represent every DMM. We also need to expand the dataset of patches that were analyzed and include the option to work with multi–threaded applications. Finally, we would like to test our framework with bigger applications (e.g., Firefox or Apache HTTPD) for patching them using our set of security patches. The biggest challenge of hot–patching the data structures of an application as complex as Firefox or Samba is that recovering data structures is not a trivial task. Therefore, we have to deal with more data structures including some that we are not able to import to our database due to their complexity. These types of data structures include: complex control flow (e.g., loops, return values) introduced into data structures, data structures modified by external libraries, and data structures that are not in the main application (e.g., kernel data structures, data structures defined in external libraries).

We concluded that, in order to minimize the complexity of hot–patching an application, patch developers should take into consideration the new semantics their patches entail. Therefore, we recommend patch–developers to minimize the control flow changes their patch introduces by using statements that we define as feasible in this thesis.

Bibliography

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

[2] LANGSEC: Language-theoretic Security, October 2013. `http://www.langsec.org/`.

[3] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus: Online Patches and Updates for Security. In *In 14th USENIX Security Symposium*, pages 287–302. USENIX Association, 2005.

[4] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 187–198, New York, NY, USA, 2009. ACM.

[5] 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 *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 26:1–26:14, Berkeley, CA, USA, 2007. USENIX Association.

[6] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing Dynamic Update in an Operating System. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 32–32, Berkeley, CA, USA, 2005. USENIX Association.

[7] Andrew R. Bernat and Barton P. Miller. Anywhere, Any-time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM.

[8] Leyla Bilge and Tudor Dumitras. Before We Knew it: An Empirical Study of Zero-Day Attacks in the Real World. In *Proceedings of the 2012 ACM Conference on Computer*

and Communications Security, CCS '12, pages 833–844, New York, NY, USA, 2012. ACM.

[9] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy Modular Upgrades in Persistent Object Stores. *SIGPLAN Not.*, 38(11):403–417, October 2003.

[10] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

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

[12] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live Updating Operating Systems Using Virtualization. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 35–44, New York, NY, USA, 2006. ACM.

[13] 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.

[14] Dustin Childs. KB2839011 Released to Address Security Bulletin Update Issue, 2013. `http://blogs.technet.com/b/msrc/archive/2013/04/11/kb2839011-released-to-address-security-bulletin-update-issue.aspx`.

[15] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious Structure Layout. *SIGPLAN Not.*, 34(5):1–12, May 1999.

[16] James C. Corbett. Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs. *ACM Trans. Softw. Eng. Methodol.*, 9(1):51–93, January 2000.

[17] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for Data Structures. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 255–266, Berkeley, CA, USA, 2008. USENIX Association.

[18] DARPA. Darpa announces cyber grand challenge. October 2013. `http://www.darpa.mil/NewsEvents/Releases/2013/10/22.aspx`.

[19] 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.

[20] DWARF Debugging Information Format Committee. Dwarf Debugging Information Format Version 4, 2010. `http://dwarfstd.org`.

[21] Daniel Fleshbourne. iPhone Users Now Fear Security Patches, Say Analysts. October 2007. `http://www.neowin.net/news/iphone-users-now-fear-security-patches-say-analysts?reply=583032`.

[22] Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer Immunology. *Communications of the ACM*, 40:88–96, 1996.

[23] Rakash Ghiya and Laurie Hendren. Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 1–15, New York, NY, USA, 1996. ACM.

[24] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A Formal Framework for On-line Software Version Change. *IEEE Trans. Software Engineering*, 22(2):120–131, February

1996. http://dx.doi.org/10.1109/32.485222.

[25] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, General-Purpose Dynamic Software Updating for C. *SIGPLAN Not.*, 47(10):249–264, October 2012.

[26] Christopher M. Hayden, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. State Transfer for Clear and Efficient Runtime Updates, 2011. http://dx.doi.org/10.1109/ICDEW.2011.5767632.

[27] Michael Hicks and Scott Nettles. Dynamic Software Updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, November 2005.

[28] Graylin Jay, Joanne E. Hale, Randy K. Smith, David P. Hale, Nicholas A. Kraft, and Charles Ward. *JSEA*, (3):137–143.

[29] Theodore Johnson and Padmashree Krishna. Lazy Updates for Distributed Search Structure. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of data*, SIGMOD '93, pages 337–346, New York, NY, USA, 1993. ACM.

[30] I.T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 1986.

[31] Alexandros Karatzoglou, David Meyer, and Kurt Hornik. Support Vector Machines in R. *Journal of Statistical Software*, 15(9):1–28, April 2006. http://www.jstatsoft.org/v15/i09.

[32] J. Zico Kolter and Marcus A. Maloof. Learning to Detect and Classify Malicious Executables in the Wild. *Journal of Machine Learning Research*, 7:2721–2744, December 2006.

[33] Evgenios Konstantinou. Metamorphic Virus: Analysis and Detection. *Masters Thesis at Royal Holloway University of London*, 2003.

[34] Kirk Lafler. In *Proc SQL: Beyond the Basics Using SAS*. SAS Publishing, 2004.

[35] Peng Li, Debin Gao, and Michael K. Reiter. Automatically Adapting a Trained Anomaly Detector to Software Patches. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 142–160, Berlin, Heidelberg, 2009. Springer-Verlag.

[36] Michael E. Locasto. Self-healing: Science, Engineering, and Fiction. In *Proceedings of the 2007 Workshop on New Security Paradigms*, NSPW '07, pages 43–48, New York, NY, USA, 2008. ACM.

[37] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[38] Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna. Protecting a Moving Target: Addressing Web Application Concept Drift. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 21–40, Berlin, Heidelberg, 2009. Springer-Verlag.

[39] Kristis Makris and Kyung Dong Ryu. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. *SIGOPS Oper. Syst. Rev.*, 41(3):327–340, March 2007.

[40] R. Manevich, T. Lev-ami, M. Sagiv, G. Ramalingam, and J. Berdine. Heap Decomposition for Concurrent Shape Analysis. In *Proceedings of the 15th international symposium on Static Analysis*, SAS '08, pages 363–377, Berlin, Heidelberg, 2008. Springer-Verlag.

[41] Elinor Mills. Symantec Pulls Norton Patch After Error Reports. *CNET News: Insecurity Complex*, August 2009. http://news.cnet.com/8301-27080_3-10317686-245.html.

[42] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical Dynamic Software Updating for C. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 72–83, New York, NY, USA, 2006. ACM.

[43] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: Automatically Correcting Memory Errors with High Probability. *Commun. ACM*, 51(12):87–95, 2008.

[44] David Oppenheimer, Aaron Brown, James Beck, Daniel Hettena, Jon Kuroda, Noah Treuhaft, David A. Patterson, and Katherine Yelick. ROC-1: Hardware Support for Recovery-Oriented Computing. *IEEE Trans. Comput.*, 51(2):100–107, February 2002.

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

[46] 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.

[47] Bogdan Popa. Windows 8 update fails on KB2770917. November 2012. http://news.softpedia.com/news/Window-8-Update-Fails-on-KB2770917-307875.shtml.

[48] 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.

[49] Ashwin Ramaswamy, Sergey Bratus, Sean W. Smith, and Michael E. Locasto. Katana: A Hot Patching Framework for ELF Executables. In *The* 4^{th} *International Workshop on Secure Software Engineering (SecSE 2010), held in conjunction with ARES 2010*, pages 507 – 512.

[50] Christopher Richardson. Virus Detection with Machine Learning. In *Master's thesis from the University of Bristol*, 2009.

[51] 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.

[52] 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 of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[53] Bernhard Schlkopf. The Kernel Trick for Distances. In *TR MSR 2000-51, Microsoft Research*, pages 5–3, 1993.

[54] Stelios Sidiroglou, Michael E. Locasto, and Angelos D. Keromytis. Hardware Support For Self-Healing Software Services. *ACM SIGARCH Computer Architecture News*, 33(1):42–47, March 2005.

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

[56] Craig A. N. Soules, Jonathan Appavoo, Dilma Da Silva, Marc Auslander, Gregory R. Ganger, and Michal Ostrowski. System Support for Online Reconfiguration. In *USENIX Annual Technique Conference*, pages 141–154, San Antonio, TX, June 2003.

[57] 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 ACM Workshop on Security and Artificial Intelligence*, AISec '09, pages 39–46, New York, NY, USA, 2009. ACM.

[58] Bjarne Steensgaard. Points-to Analysis by Type Inference of Programs with Structures and Unions. In *6th International Conference, Compiler Construction '96*, pages 136–150. Springer-Verlag, 1996.

[59] 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.

[60] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic Software Updates: a VM-Centric Approach. *SIGPLAN Not.*, 44(6):1–12, June 2009.

[61] Symantec. Targeted Attack Exploits Ichitaro Vulnerability. June 2013. `http://www.symantec.com/connect/blogs/targeted-attack-exploits-ichitaro-vulnerability-0`.

[62] Tool Interface Standard TIS. Executable and Linkable Format Specification. 1993. `http://www.skyfree.org/linux/references/ELF_Format.pdf`.

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

[64] CVE Official Website. Common Vulnerabilities and Exposures (CVE), http://cve.mitre.org/.

[65] Westley Weimer, Than Vu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[66] DWARF Debugging Information Format Workgroup. Dwarf debugging standard. 2013. `http://dwarfstd.org/`.

[67] 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*, pages 221–229, 1982.