© 2011 by Anthony Edward Cozzie. All rights reserved.

DETECTING AND COMBINING PROGRAMMING PATTERNS

BY

ANTHONY EDWARD COZZIE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Assistant Professor Samuel King, Chair
Professor Remzi Arpaci-Dusseau, University of Wisconsin at Madison
Associate Professor Eyal Amir
Associate Professor Vikram Adve
Assistant Professor Darko Marinov

Abstract

This thesis explores detecting patterns in the most general interface to computers: source and assembly program code. Because writing computer programs correctly is so difficult, there is a large assortment of software engineering techniques devoted to making this process easier and more efficient. Therefore, despite the huge space of possible programs, most programs written by humans will exhibit some of the same patterns.

Project Laika detects the data structures used by guest programs using unsupervised Bayesian learning. Using these data structures, it can detect viruses with considerably better accuracy than ClamAV, a leading industry solution. Using high-level features makes Laika considerably more resistant to polymorphic worms, because it requires them to preserve their high-level structure through their polymorphic transformations.

Project Macho generates Java programs from a combination of natural language, example inputs, and a database of Java code. Unlike past natural language programming systems, which were basically verbose versions of Visual Basic, Macho allows users to specify (small) programs in high-level language and use examples to fill in the details. We were able to generate satisfactory solutions for basic versions of several core utilities, including ls and grep, even when the natural language inputs were varied substantially.

To humanity.

Acknowledgments

No man is an island. This work would not have been possible without the support and advice of my advisor, thesis committee, and research group, who not only read my papers but also played on my basketball team.

Mark Grechanik donated a large database of Java code to the Macho project.

Table of Contents

List of Tables . . . . **vii**

List of Figures . . . . **viii**

Chapter 1 Introduction **1**
1.1 Thesis Statement 2
1.2 The Road Ahead 2

Chapter 2 Painless Introduction to Machine Learning **3**
2.1 Methodology 3
2.2 Conditional Probability and Bayes' Rule 4
2.3 Classification 5
2.4 Natural Language Processing 6

Chapter 3 Digging for Data Structures **8**
3.1 Introduction 8
3.2 Data Structure Detection 10
3.2.1 Atoms and Block Types 11
3.2.2 Finding Data Structures 12
3.2.3 Exploiting Malloc 12
3.2.4 Bayesian Model 12
3.2.5 Typed Pointers 13
3.2.6 Dynamically-Sized Arrays 14
3.3 Implementation 14
3.4 Data Structure Detection Results 15
3.5 Program Detection 18
3.5.1 Discussion 20
3.5.2 Analysis 20
3.5.3 Scaling 22
3.6 Related Work 22
3.6.1 The Unobfuscated Semantic Gap 23
3.6.2 The Obfuscated Semantic Gap 23
3.6.3 The Deobfuscated Semantic Gap 23
3.6.4 Shape Analysis 24
3.6.5 Reverse Engineering 24
3.7 Conclusions 25

Chapter 4 Macho: Programming with Man Pages **26**
4.1 Introduction 26
4.2 Design and Implementation 28
4.2.1 Natural Language Parser 28
4.2.2 Database 30

4.2.3 Stitching . . . 31
4.2.4 Automated Debugger . . . 32
4.3 Evaluation . . . 32
4.3.1 pwd . . . 35
4.3.2 cat . . . 35
4.3.3 sort . . . 35
4.3.4 cp . . . 36
4.3.5 wget . . . 36
4.3.6 grep . . . 36
4.3.7 ls . . . 37
4.3.8 Head and Uniq . . . 37
4.4 Lessons Learned . . . 39
4.4.1 The Database is King . . . 39
4.4.2 Pure NLP is Bad . . . 39
4.4.3 Interactive Programming is the Answer . . . 40
4.4.4 Abstractions are Key . . . 40
4.5 Related work . . . 40
4.5.1 Natural-Language Processing . . . 40
4.5.2 Natural Language Programming . . . 41
4.5.3 Programming by example . . . 42
4.5.4 Code snippet search . . . 42
4.5.5 Programming by sketching . . . 42
4.5.6 Deductive program synthesis . . . 42
4.6 Conclusions . . . 43

Chapter 5 Conclusions . . . 44

Appendix Raw Code for Macho Programs . . . 45

References . . . 54

Author's Biography . . . 58

List of Tables

3.1 Terms and symbols 14
3.2 Data Structure Detection Accuracy. The first part of the table shows Laika's accuracy using only the memory image; the second part using the memory image and a list of the sizes and locations of objects. $p_{obj}(real|laika)$ and $p_{obj}(laika|real)$ are the accuracy when known atomic types like *int* or *char* are ignored. 16
3.3 Classification Results. For example, we used 17 of our 34 Kraken samples as training data. When the remaining 16 samples were compared against the signature, they produced an average mixture ratio of 0.52 with a standard deviation of 0.021. Of our 27 clean images, we used 13 as training data, and those produced an average mixture ratio with our Kraken sample of 0.83 with a standard deviation of 0.078. The resulting maximum likelihood classifier classifies anything with a mixture ratio of less than 0.58 as Kraken, and has an estimated accuracy of 99.8%; it classified the remaining 17 Kraken samples and 14 standard Windows applications without error. If a new sample was compared with the Kraken signature and produced a mixture rate of 0.56, it would be classified as Kraken, being 1.9σ from the average of the Kraken samples and 3.5σ from the average of the normal samples. 18

List of Figures

3.1 An example of data structure detection. On the left is a small segment of the heap, and on the right is Laika's output. Class 1, in rows 2-13, is a doubly linked list of C strings; the first two elements are pointers to other elements of Class 1. The fourth element is actually internal malloc data on the size of the next chunk. Laika estimates the start of the next object as the end of the first, which is 8 bytes too long. Class 2 contains two C null-terminated character arrays. A real heap sample is much noisier; in the programs we looked at, less than 50% of the heap was occupied by active objects; the rest was a mix of freed objects, malloc padding, and unallocated chunks. 10
3.2 Laika generated type graph for Privoxy 17
3.3 Correct type graph for Privoxy 17

4.1 Macho workflow 27
4.2 Examples of patterns mined by the various databases. In each case, a pattern is a block of code (a single function in the first database, a full Java expression in the third) which matches the names expressed in a query. All of the functions in this figure match the pattern 'has an input variable named directory and an output variable named files'. 29
4.3 Macho's results for generating select core utils. This figure shows the results for pwd, cat, sort, grep, ls, cp, wget, head, and uniq, the natural language input we used for each of these programs, and a very short description of why Macho succeeded or failed. We discuss each example in detail in the text. . . 33
4.4 This figure shows the number of stitching candidates that the automated debugger had to examine before it could find one it could patch up to pass the unit test. It shows that Macho is not just flailing randomly - many programs are solved with relatively few stitching candidates. Interestingly the most difficult program to synthesize was actually pwd. 33
4.5 This figure shows the percentage of lines of code correct of Macho's *best* solution after stitching and synthesis (the average number of lines correct for a stitching solution would be considerably lower) as determined by me. For the most part Macho either succeeds beautifully or fails horribly, which explains the sharp phase transition. I selected lines of code rather than comparing the outputs because even a few lines of erroneous code are usually enough to completely alter the output. For details on which programs were partially successful, see the appendix. 34
4.6 Final solution for ls. 38
4.7 Final solution for ls. 39

A.1 Code for Head 45
A.2 Code for Cp 46
A.3 Code for Sort 47
A.4 Code for Pwd 48
A.5 Code for Cat 49
A.6 Code for Grep 50
A.7 Code for ls-1 51
A.8 Code for ls-2 52
A.9 Code for Wget 52
A.10 Code for Uniq 53

Chapter 1

Introduction

System designers use abstractions to make building complex systems easier. Fixed interfaces between components allow their designers to innovate separately, reduce errors, and construct the complex computer systems we use today. The best interfaces provide exactly the right amount of detail, while hiding most of the implementation complexity.

However, no interface is perfect. When system designers need additional information they are forced to bridge the gap between levels of abstraction. The easiest, but most brittle, method is to simply hard-code the mapping between the interface and the structure built on top of it. Hard-coded mappings enable virtual machine monitor based intrusion detection [23, 36] and discovery of kernel-based rootkits using a snapshot of the system memory image [52]. More complicated but potentially more robust techniques infer details by combining general knowledge of common implementations and runtime probes. These techniques allow detection of OS-level processes from a VMM using CPU-level events [32, 34], file-system-aware storage systems [33, 58], and storage-aware file systems [50]. Most of these techniques work because the interfaces they exploit can only be used in a very limited number of ways. For example, only an extremely creative engineer would use the CR3 register in an x86 processor for anything other than process page tables.

This thesis explores detecting patterns in the most general interface to computers: source and assembly program code. Because writing computer programs correctly is so difficult, there is a large assortment of software engineering techniques devoted to making this process easier and more efficient. Ultimately most of these techniques revolve around the same ideas of abstraction and divide-and-conquer as the original interfaces. Whether this is the only way to create complex systems remains to be seen, but in practice these ideas are pounded into prospective programmers by almost every text on computer science, from *The Art of Computer Programming* to the more bourgeoisie *Visual Basic for Dummies*.

Therefore, despite the huge space of possible programs, most programs written by humans will exhibit some of the same patterns. For example, most programs organize their memory into objects, or data structures, blocks of contiguous memory which store related values. Alternatively, many programmers use libraries, and often learn how to use these libraries via standard examples distributed with the language, leading to similar patterns.

1.1 Thesis Statement

We can build simple machine learning models for how humans write code, and use them to detect data structures and combine code snippets into small programs.

1.2 The Road Ahead

The rest of this thesis contains a very simple introduction to the basics of machine learning for systems researchers, and implementations of two simple systems based on those principles.

Project Laika detects the data structures used by guest programs using unsupervised Bayesian learning. Because each program will use some tens or hundreds unique data structures, it is difficult to represent them with a simple model, making machine learning a fast alternative to months of human reverse engineering. Laika leverages pointers: machine words that point to other data structures will have significantly different byte values than machine words which contain integers or strings. Full data structure layouts are reconstructed from these features upward.

Using these data structures, Laika can detect viruses with considerably better accuracy than ClamAV, a leading industry solution. Using high-level features makes Laika considerably more resistant to polymorphic worms, because it requires them to preserve their high-level structure through their polymorphic transformations.

Project Macho generates Java programs from a combination of natural language, example inputs, and a database of Java code. Unlike past natural language programming systems, which were basically verbose versions of Visual Basic, Macho allows users to specify (small) programs in high-level language and use examples to fill in the details. In other words, Macho allows users to be concrete and precise without being formal.

The primary problem with natural language programming is its ambiguity, which creates difficulties both for the system, which must guess at a correct solution, and for the user, who does not know which solution was chosen or whether the request was even reasonable. Macho solves the first problem by looking up code snippets in a database of code, rather than small manually defined rule sets, and the second by comparing the output of candidate programs with the example output. Macho was able to generate satisfactory solutions for basic versions of several core utilities, including ls and grep.

Chapter 2

Painless Introduction to Machine Learning

Machine learning is the construction of mathematical models of reality directly from observations. This is most useful when our model consists of simple relations but many parameters. Fortunately, the mathematics behind basic machine learning (which are all that I have used in this thesis) are actually quite simple.

2.1 Methodology

Science works by modeling life with mathematics. The fact that this works at all is actually quite surprising. Suppose you plan to drive from Chicago to New York. If you know the distance is 800 miles, and you estimate your speed on the interstate at 65 miles per hour, it will take about 12 hours. The fact that we can model the zillions of particles that make up the road between Chicago and New York, the cars that drive on it, the people that control them, and the air the floats on it, with a line segment and a point moving at constant velocity is nothing short of amazing. And in fact you might have a flat tire, hit road construction, or, if you are a lovelorn graduate student, fall in love with a hooker at a rest stop in Ohio.

Scientists make progress by generating models (theories) which more accurately model physical reality. Google Maps models the roads between Chicago and New York as a graph with edge weights that represent the average time a car requires to traverse them. This gives a somewhat more accurate estimate of 13.8 hours, but it is far from perfect, as Google Maps does not consider the condition of your car, or what traffic will be like in Manhattan when you arrive, or whether a butterfly will flap its wings in Dallas and produce tornadoes in Illinois, all of which might effect the driving time. Instead, the designers pray these unruly terms will be small and group them under the term “error”.

The magic of models is that we can use mathematics to turn the crank and move into the future. If we know a particle is moving along a line at a constant velocity, we can predict where it will be at any time in the future, and even prove simple properties about it. Knowing whether things will work ahead of time is obviously extremely valuable, and is the basis for all technology since the wheel. The scientific method simply directs the scientist to pick models that work empirically, something that should be obvious, but is actually rather difficult for humans who evolved to make quick decisions. The caveman who reasoned ’Even though the tiger ate my friend, it’s probably just a coincidence’

would not be part of the gene pool for long.

Some problems are much easier than others. We can roughly divide our problems into three classes: simple, complex, and chaotic. Simple systems (like gravity) can represent huge numbers of particles with just a few variables and simple relations. Complex systems (like natural language or face recognition) require more complicated mathematics, but usually generate more or less correct results. Chaotic systems (like the weather) may be simple at a very low level, but require huge numbers of parameters and are inherently very difficult to model. Generally speaking, physics deals with the most simple and accurate models, then chemistry, then biology, psychology, and finally, at more or less the voodoo level, economics and social sciences [17].

Most of the time, the models are generated by humans. Legend has Newton sitting under an apple tree when he decided to model gravity, Einstein thinking about traveling alongside a wave of light, and, based on my experience in New York, the Google Engineers were probably playing foosball and drinking soda. A well trained engineer can examine the problem, compare it with his experience, and suggest a simple representation. The alternative is to do so mechanically using the mathematics of machine learning. The process is identical: make observations and compute the model that best fits the observations. Since there is an infinite space of models and features, usually a human is required to select features and simple relations.

Machine learning works best for complex systems. Chaotic systems are almost impossible to predict, and simple systems can be accurately solved without using machine learning. As previously stated, the goal of modeling the world using mathematics is to predict the future. In machine learning, this is accomplished by measuring the correlations between known variables (features or inputs) and output variables. This process is identical for information and coding theory (predicting the remainder of a message given part of it), classification (predicting the true class of an object given its features), or simple regression (predicting how one quantity will change in response to another).

2.2 Conditional Probability and Bayes' Rule

Since our goal is to build a predictor, we are interested in the probability of a certain outcome given our features, the things we can know beforehand. In the language of probability, this is written:

$$P(outcome|features) \tag{2.1}$$

In other words, given that a fish weighs 50 lbs, what is the probability that it is a salmon? This distribution is usually calculated using Bayes' rule, which states that:

$$P(outcome|features) = \frac{P(features|outcome)P(outcome)}{P(features)} \tag{2.2}$$

because it is easy to calculate $P(features|outcome)$ from our training data: just estimate the distribution from the features of the samples labeled with a given class (catch a bunch of salmon, and measure how many of them weigh 50 lbs. Who said science can't be tasty?). You can think of Bayes' rule as splitting the evidence we have into before and after the observation. Although the mathematics is simple, Bayes' rule is actually quite deep. Many people struggle with the intuition behind reversing conditional probability, as we can show with a classic example.

Siblings

Given that a family has two children and that one of them is a boy, what is the probability that the other child is a boy? A quick glance to Bayes' rule gives us:

$$P(twoboys|oneboy) = \frac{P(oneboy|twoboys)P(twoboys)}{P(oneboy)} = \frac{1.0 * 0.25}{0.75} = \frac{1}{3} \tag{2.3}$$

Which is a rather unintuitive result. The easiest way to think of this is that there are four possibilities, BB, BG, GB, and GG, one of which (GG) is ruled out. However, we can restore normalcy to the universe by changing our question ever so slightly: *What is the probability that a man has a brother, assuming all parents have two children?*

$$P(brother|man) = \frac{P(man|brother)P(brother)}{P(man)} = \frac{0.5 * 0.5}{0.5} = \frac{1}{2} \tag{2.4}$$

So by changing the sampling from family to child our answer changes significantly, the easiest way to think of this being that BB now counts twice.

2.3 Classification

Given the correct conditional probabilities, predicting the future is simple: pick the most likely outcome. If we want to decide whether our 50 lb fish is a salmon or a seabass, we simply test:

$$P(salmon|50lbs) > P(seabass|50lbs) \tag{2.5}$$

Which we can calculate from training data using Bayes' rule as described above. However, we usually have more than one piece of information. In addition to the weight, we might have scale color, eye color, latitude and longitude of where the fish was caught, and so on. This requires the joint distribution:

$$P(salmon|50lbs, silverscales, redeyes, caught@47.45, 123.15) \tag{2.6}$$

Which is simply our estimate for whether a fish is a salmon given all of our evidence. Unfortunately, the number of samples needed to estimate the joint distribution grows exponentially in the number of features, so the usual method is to assume that features are independent. This is almost always a terrible assumption. If salmon are present in Seattle, there is no guarantee they will be present in North Dakota, at roughly the same latitude, or that salmon that weigh 50 lbs will have the same scale color as 1 lb fry. However, it dramatically simplifies our mathematics, because if A and B are independent:

$$P(A,B) = P(A)P(B) \tag{2.7}$$

Given this assumption, we can classify whether our fish is a salmon or seabass relatively easily. We want to test whether:

$$\frac{P(salmon|features)}{P(seabass|features)} > 1 \tag{2.8}$$

$$\frac{P(features|salmon)P(salmon)}{P(features|seabass)P(seabass)} > 1 \tag{2.9}$$

$$ln(P(salmon)) - ln(P(seabass)) + ln(P(50lbs|salmon)) - ln(P(50lbs|salmon) + ... > 0 \tag{2.10}$$

because this equation is a linear sum of terms, it implies that our decision region is a hyperplane in the space of the features. As expected, it will fail completely when its assumptions are violated and the inputs are correlated. Imagine we are trying to estimate $P(checksum|bit1, bit2)$ where our checksum is an XOR parity bit - there is no line which can separate the two regions. In practice, naive Bayesian classifiers work reasonably well for most problems and are more or less the default. For more difficult problems, the engineer can use neural networks or support vector machines. The other main trick is to project the features into a different space. For example, the xor classifier will have great performance if we try to classify based on F(bit1, bit2) where F(x,y) happens to be XOR.

2.4 Natural Language Processing

While Laika uses unsupervised Bayesian learning, Macho uses off the shelf natural language processing tools, specifically a part of speech tagger, which attempts to determine the grammatical part of speech for each word, e.g. noun or verb, and which pieces are connected. Essentially it diagrams the sentence.

This is a fairly challenging task because many words in English have multiple meanings and can be used in multiple parts of speech. Consider the different meanings of the word set in these sentences:

* What a beautiful set of silverware.

* That last set of deadlifts killed me.

* Osiris and Set demanded human sacrifice on Tuesdays.

* Turn off the TV set you ingrate!

* He set the apple on the table.

* The sun set brilliantly yesterday.

* I'm going to leave the pie on the porch to set.

* We must set the line here, no farther!

* The warrior was set and ready, but the gargoyle was set in stone.

This has created a rich literature which is far to extensive to seriously cover here; researchers have tried Markov models, neural networks, context free grammars, and almost every other method imaginable. The key problem is the size of the feature space (words depend on each other) and the complexity and ambiguity of natural language and its dependence on common sense reasoning, a problem which is central to Macho.

Chapter 3

Digging for Data Structures

3.1 Introduction

System designers use abstractions to make building complex systems easier. Fixed interfaces between components allow their designers to innovate separately, reduce errors, and construct the complex computer systems we use today. The best interfaces provide exactly the right amount of detail, while hiding most of the implementation complexity.

However, no interface is perfect. When system designers need additional information they are forced to bridge the gap between levels of abstraction. The easiest, but most brittle, method is to simply hard-code the mapping between the interface and the structure built on top of it. Hard-coded mappings enable virtual machine monitor based intrusion detection [23, 36] and discovery of kernel-based rootkits using a snapshot of the system memory image [52]. More complicated but potentially more robust techniques infer details by combining general knowledge of common implementations and runtime probes. These techniques allow detection of OS-level processes from a VMM using CPU-level events [32, 34], file-system-aware storage systems [33, 58], and storage-aware file systems [50]. Most of these techniques work because the interfaces they exploit can only be used in a very limited number of ways. For example, only an extremely creative engineer would use the CR3 register in an x86 processor for anything other than process page tables.

The key contribution of this paper is the observation that even more general interfaces are used often by programmers in standard ways. Because writing computer programs correctly is so difficult, there is a large assortment of software engineering techniques devoted to making this process easier and more efficient. Ultimately most of these techniques revolve around the same ideas of abstraction and divide-and-conquer as the original interfaces. Whether this is the only way to create complex systems remains to be seen, but in practice these ideas are pounded into prospective programmers by almost every text on computer science, from *The Art of Computer Programming* to the more bourgeoisie *Visual Basic for Dummies*.

We chose to exploit a small piece of this software engineering panoply, the compound data structure. Organizing data into objects is so critical for encapsulation and abstraction that even programmers who do not worship at the altar of object-oriented programming usually use a significant number of data structures, if only to implement abstract data

types like trees and linked lists. Therefore we can expect the memory image of a process to consist of a large number of instantiations of a relatively small number of templates.

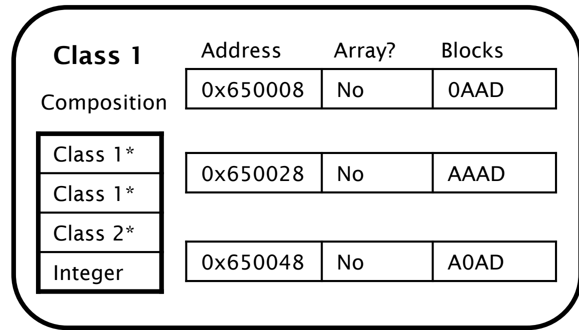
This paper describes the design and implementation of a system – which we named Laika in honor of the Russian space dog – for detecting those data structures given a memory image of the program. The two key challenges are identifying the positions and sizes of objects, and determining which objects are similar based on their byte values. We identify object positions and sizes by using potential pointers in the image to estimate object positions and sizes. We determine object similarity by converting objects from sequences of raw bytes into sequences of semantically valued blocks: "probable pointer blocks" for values that point into the heap or the stack, "probable string blocks" for blocks that contain null-terminated ASCII strings, and so on. Then, we cluster objects with similar sequences of blocks together using Bayesian unsupervised learning.

Although conceptually simple, detecting data structures in practice is a difficult machine learning problem. Because we are attempting to detect data structures without prior knowledge, we must use unsupervised learning algorithms. These are much more computationally complex and less accurate than supervised learning algorithms that can rely on training data. Worse, the memory image of a process is fairly chaotic. Many malloc implementations store chunk information inside the chunks, blending it with the data of the program. The heap is also fairly noisy: a large fraction consists of effectively random bytes, either freed blocks, uninitialized structures, or malloc padding. Even the byte/block transformation is error-prone, since integers may have values that "point" into the heap. Despite these difficulties, Laika manages reasonable results in practice.

To demonstrate the utility of Laika, we built a virus checker on top of it. Current virus checkers are basically sophisticated versions of grep [2]. Each virus is identified with a fingerprint, usually a small sequence of instructions. When the virus checker finds that fingerprint in a program, it classifies it as a version of the corresponding virus. Because it is easy to modify the instruction stream of a program in provably correct ways, virus writers have created polymorphic engines that replace one set of instructions with another computationally equivalent one, obfuscating the fingerprints [40]. Most proposals to combat polymorphic viruses have focused on transforming candidate programs into various canonical formats in order to run fingerprint scanners [8, 13, 38].

Instead, our algorithm classifies programs based on their data structures: if an unknown program uses the same data structures as Agobot, it is likely to in fact be a copy of Agobot. Not only does this bypass all of the code polymorphism in current worms, but the data structures of a program are likely to be considerably more difficult to obfuscate than the executable code – roughly compiler-level transformations, rather than assembler-level ones. Our polymorphic virus detector based on Laika is over 99% accurate, while ClamAV, a leading open source virus detector, manages only 85%. Finally, by detecting programs based on completely different features our detector has a strong synergy with traditional code-based virus detectors.

Address	Value	Char Value	Block
0x650000	0x20	“!”	**D**
0x650008	0x0	“\0”	**0**
0x650010	0x650028	“\FS\0e”	**A**
0x650018	0x650088	“\^\0e”	**A**
0x650020	0x20	“!”	**D**
0x650028	0x650008	“\BS\0e”	**A**
0x650030	0x650048	“0\0e”	**A**
0x650038	0x650068	“h\0e”	**A**
0x650040	0x20	“!”	**D**
0x650048	0x650028	“\FS\0\e”	**A**
0x650050	0x0	“\0”	**0**
0x650058	0x650068	“h\0e”	**A**
0x650060	0x20	“!”	**D**
0x650068	0x6873696620656E6F	“one fish”	**S**
0x650070	0x6966206F7774202C	“, two fi”	**S**
0x650078	0x20646572202C6873	“sh, red ”	**S**
0x650080	0x20	“!”	**D**
0x650088	0x6C62202C68736966	“fish, bl”	**S**
0x650090	0x2E68736966206575	“ue fish.”	**S**
0x650098	0x56700	“\0g\ENQ”	**D**
0x6500A0	0x40	“A”	**D**



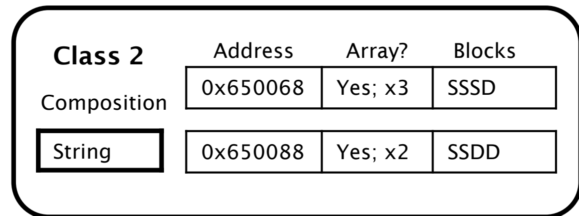


Figure 3.1: An example of data structure detection. On the left is a small segment of the heap, and on the right is Laika's output. Class 1, in rows 2-13, is a doubly linked list of C strings; the first two elements are pointers to other elements of Class 1. The fourth element is actually internal malloc data on the size of the next chunk. Laika estimates the start of the next object as the end of the first, which is 8 bytes too long. Class 2 contains two C null-terminated character arrays. A real heap sample is much noisier; in the programs we looked at, less than 50% of the heap was occupied by active objects; the rest was a mix of freed objects, malloc padding, and unallocated chunks.

Memory-based virus detection is especially effective now that malware writers are turning from pure mayhem to a greedier strategy of exploitation [1, 22]. A worm that merely replicates itself can be made very simple, to the point that it probably does not use a heap, but a botnet that runs on an infected computer and provides useful (at least to the botnet author) services like DoS or spam forwarding is more complex, and more complexity means more data structures.

3.2 Data Structure Detection

A classifier is an algorithm that groups unknown objects, represented by vectors of features, into semantic classes. Ideally, a classification algorithm is given both a set of correctly classified training data and a list of features. For example, to classify fruit the algorithm might be given the color, weight, and shape of a group of oranges, apples, watermelons, and bananas, and then asked whether a 0.1 kg red round fruit is an apple or a banana. This is called supervised learning; a simple example is the naive Bayes classifier, which learns a probability distribution for each feature for each class from the training data. It then computes the class membership probability for unknown objects

as the product of the class feature probabilities and the prior probabilities, and places the object in the most likely class. When we do not have training data, we must fall back to unsupervised learning. In unsupervised learning, the algorithm is given a list of objects and features and directed to create its own classes. Given a basket of fruit, it might sort them into round orange things, round red things, big green things, and long yellow things. Designing a classifier involves selecting features that expose the differences between items and algorithms that mirror the structure of the problem.

3.2.1 Atoms and Block Types

The most important part of designing any classifier is usually selecting the features. Color, shape, and weight will work well for fruit regardless of what algorithm is used, but country of origin will not. This problem is particularly acute for data structure detection, because two objects from the same class may have completely different byte values. Our algorithm converts each machine word (4 on 32-bit machines, 8 bytes on 64-bit machines) into a *block type*. The basic block types are address (points into heap/stack), zero, string, and data (everything else). This converts objects from vectors of bytes into vectors of block types, and we can expect the block type vectors to be similar for objects from the same class.

Classes are represented as vectors of *atomic types*. Each atomic type roughly corresponds to one block type (e.g., pointer → address and integer → data), but there is some margin for error since the block type classification is not always accurate; some integer or string values may point into the heap, some pointers may be uninitialized, a programmer may have used a union, and so on. This leads to a probability array $p(blocktype|atomictype)$ where the largest terms are on the diagonal, but all elements are nonzero. While these probabilities will not be exactly identical for individual applications, in our experience they are similar enough that a single probability matrix suffices for most programs. In the evaluation we measured $p(blocktype|atomictype)$ for several programs; all of them had strongly diagonal matrices. There is also a random atomic type for blocks that lie between objects. Figure 4.5 shows an example of what memory looks like when mapped to block and atomic types.

Although the block type/atomic type system allows us to make sense of the otherwise mystifying bytes of a memory image, it does have some problems. It will miss unaligned pointers completely, since the two halves of the pointer are unlikely to point at anything useful, and it will also miss the difference between groups of integers, for example four 2-byte integers vs. one 8-byte integer. In our opinion these problems are relatively minor, since unaligned pointers are quite rare, and it would be extremely difficult to distinguish between four short ints and one long int anyway. Lastly, if the program occupies a significant fraction of its address space there will be many integer/pointer mismatches, but as almost all new CPUs are 64-bit, the increased size of a 64-bit address space will reduce the probability of false pointers.

3.2.2 Finding Data Structures

Unlike the idyllic basket of fruit, it is not immediately obvious where the objects lurk in an image, but we can estimate their positions using the targets of the pointers. Our algorithm scans through a memory image looking for all pointers, and then tentatively estimates the positions of objects as those addresses referenced in other places of the image.

Although pointers can mark the start of an object, they seldom mark the end. Laika estimates the sizes of objects during clustering. Each object's size is bounded by the distance to the next object in the address space, and each class has a fixed size, which is no larger than its smallest object. If an object is larger than the size of its class, its remaining blocks are classified as random noise. For this purpose we introduce the random atomic type, which generates all block types more or less equally. In practice some objects might be split in two by internal pointers - pointers to the interior of malloc regions. At the moment we do not merge these objects.

3.2.3 Exploiting Malloc

It should be possible for Laika to find objects more accurately when armed with prior information about the details of the malloc library. For example, the Lea allocator, used in GNU libc, keeps the chunk size field inlined at the top of the chunk as shown in figure 4.5. Unfortunately there is sufficient variety in malloc implementations to make this approach tedious. Aside from separate malloc implementations for Windows and Linux, there are many custom allocators, both for performance and other motivations like debugging.

Early versions of Laika attempted to exploit something that most malloc implementations should share: address space locality. Most malloc implementations divide memory into chunks, and chunks of the same size often lie in contiguous regions. Since programs exhibit spatial locality as well, this means that an object will often be close to one or more objects of the same type in memory; in the applications we measured over 95% of objects had at least one object of the same class within their 10 closest neighbors. Despite this encouraging statistic, the malloc information did not significantly change Laika's classification accuracy. We believe this occurs because Laika already knows that nearby objects are similar due to similar size estimations.

3.2.4 Bayesian Model

Bayesian unsupervised learning algorithms compute a joint probability over the classes and the classification, and then select the most likely solution. We represent the memory image by $\boldsymbol{M}$, where $\boldsymbol{M}_l$ is the lth machine word of the memory image, $\boldsymbol{\omega}$ for the list of classes, where $\boldsymbol{\omega}_{jk}$ is the kth atomic type of class j, and $\boldsymbol{X}$ for the list of objects, where $\boldsymbol{X}_i$ is the position of the ith object. We do not store the lengths of objects, since they are implied by the classification. Our notation is summarized in Table 3.1.

We wish to estimate the most likely set of objects and classes given a particular memory image. With Bayes's rule, this gives us:

$$p(\boldsymbol{\omega}, \boldsymbol{X}|\boldsymbol{M}) = \frac{p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})p(\boldsymbol{X}|\boldsymbol{\omega})p(\boldsymbol{\omega})}{p(\boldsymbol{M})} \tag{3.1}$$

Since Laika is attempting to maximize $p(\boldsymbol{\omega}, \boldsymbol{X}|\boldsymbol{M})$, we can drop the normalizing term $p(\boldsymbol{M})$, which does not affect the optimum solution, and focus on the numerator. $p(\boldsymbol{\omega})$ is the prior probability of the class structure. To simplify the mathematics, we assume independence both between and within classes, even though this assumption is not really accurate (classes can often contain arrays of basic types or even other objects). We let $\boldsymbol{\omega}_{jk}$ be the kth element of class j, which lets us simply multiply out over all such elements:

$$p(\boldsymbol{\omega}) = \prod_j \prod_k p(\boldsymbol{\omega}_{jk}) \tag{3.2}$$

$p(\boldsymbol{X}|\boldsymbol{\omega})$ is the probability of the locations and sizes of the list of objects, based on our class model and what we know about data structures. This term is 0 for illegal solutions where two objects overlap, and 1 otherwise. $p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})$ represents how well the model fits the data. When the class model and the instantiation list are merged, they predict a set of atomic data types (including the random "type") that cover the entire image. Since we know the real block type $\boldsymbol{M}_l$, we can compute the probability of each block given classified atomic type:

$$p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X}) = \prod_l p(\boldsymbol{M}_l|\boldsymbol{\omega}, \boldsymbol{X}) \tag{3.3}$$

When $p(\boldsymbol{\omega})$, $p(\boldsymbol{X}|\boldsymbol{\omega})$, and $p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})$ are multiplied together, we finally get a master equation which we can use to evaluate the likelihood of a given solution. Although the master equation is somewhat formidable, the intuition is very simple; $p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})$ penalizes Laika whenever it places an object into an unlikely class, thus ensuring that the solution reflects the particular memory image, while $p(\boldsymbol{\omega})$ enforces Occam's razor by penalizing Laika whenever it creates an additional class, thus causing it to prefer simpler solutions. Chapter 2 contains a simple explanation of Naive Bayesian classifiers.

3.2.5 Typed Pointers

While the simple pointer/integer classification system already produces reasonable results, a key optimization is the introduction of typed pointers. If all of the instances of a class have a pointer at a certain offset, it is probable that the targets of those pointers are also in the same class. As the clustering proceeds and the algorithm becomes more confident of the correct clustering, it changes the address blocks to typed address blocks based on the class of their

atomic type block type data structure	A machine level type, like a pointer. Value of an atomic type "struct 1". Compound type.
$\boldsymbol{X}_i$	instantiation / object i
i	index of objects
$\boldsymbol{\omega}_j$	class / compound data type j
j	index of classes
k	block offset within a class /object
$\boldsymbol{M}$	the memory image of our process
l	index with a memory image

Table 3.1: Terms and symbols

target. Typed pointers are especially important for small objects, because the class is smaller and inherently less descriptive. They also allow Laika to classify objects that contain no pointers, which can sometimes be accurately grouped solely by their references. Since it is impossible to measure the prior and posterior probabilities for the classes and pointers of an unknown program, we simply measured the probability that a typed pointer referred to a correctly typed address or an incorrectly typed address. Typed pointers greatly increase the computational complexity of the equation, because the classification of individual objects is no longer independent if one contains a pointer to the other. Worse, when Laika makes a mistake, the typed pointers will cause this error to propagate. Since typed pointer mismatches are weighted very heavily in the master equation, Laika may split the classes that reference the poorly classified data structure as well.

3.2.6 Dynamically-Sized Arrays

The second small speed bump is the dynamically-sized array. In standard classification, all elements in a class have feature vectors of the same size. Obviously this is not true with data structures, with the most obvious and important being the ubiquitous C string. We handle a dynamic array by allowing objects to "wrap around" modulo the size of a class. In other words, we allow an object to be classified as a contiguous set of instantiations of a given class - an array.

3.3 Implementation

We implemented Laika in Lisp; the program and its testing tools total about 5000 lines, including whitespace and comments. The program attempts to find good solutions to the master equation when given program images.

Unfortunately our master equation is computationally messy. Usually, unsupervised learning is difficult, while supervised learning is simple: each item is compared against each class and placed in the class that gives the least error. But with our model, if $\boldsymbol{X}_1$ contains a pointer to $\boldsymbol{X}_2$, the type of $\boldsymbol{X}_2$ will affect the block type and therefore the

classification error of $\boldsymbol{X}_1$, so even an exact supervised solution is difficult. Therefore our only choice is to rely on an approximation scheme. Laika computes $p(\boldsymbol{\omega}, \boldsymbol{X}|\boldsymbol{M})$ incrementally and uses heuristics to decide which classification changes (e.g., move $\boldsymbol{X}_{233}$ from $\boldsymbol{\omega}_{17}$ to $\boldsymbol{\omega}_{63}$). We leverage typed pointers to compute reasonable changes. For example, whenever an object is added to a class that contains a typed pointer, it tries to move the pointer targets of that object into the appropriate class.

We would like to compute the normalizing term $p(\boldsymbol{M})$ in order to obtain the actual probability of a solution, and therefore an estimate of its quality, but this would be very expensive because $p(\boldsymbol{M})$ is the sum of the probability of all possible solutions:

$$p(\boldsymbol{M}) = \sum_{\boldsymbol{X},\boldsymbol{\omega}} p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})p(\boldsymbol{X}|\boldsymbol{\omega})p(\boldsymbol{\omega}) \tag{3.4}$$

Therefore Laika estimates $p(\boldsymbol{M})$. The intuition is that $p(\boldsymbol{M})$ will be dominated by $p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})$ and especially by those solutions that pair most of the blocks with their corresponding atomic type. The estimate simply ignores the $p(\boldsymbol{X}|\boldsymbol{\omega})$ and $p(\boldsymbol{\omega})$ terms.

$$\hat{p}(\boldsymbol{M}) = \prod_{l} \arg\max_{a} p(\boldsymbol{M}_l|a) \tag{3.5}$$

Laika can then estimate the quality of each class based on the ratio of its contributions to the master equation and the estimate $\hat{p}(\boldsymbol{M})$.

3.4 Data Structure Detection Results

Measuring Laika's ability to successfully identify data structures proved surprisingly difficult. Because the programs we measured are not typesafe, there is no way to determine with perfect accuracy the types of individual bytes. The problems begin with unions, continue with strange pointer accesses, and climax with bugs and buffer overflows. Fortunately these are not too common, and we believe our ground truth results are mostly correct.

We used Gentoo Linux to build a complete set of applications and libraries compiled with debugging symbols and minimal optimizations. We then ran our test programs with a small wrapper for *malloc* which recorded the backtrace of each allocation, and used GDB to obtain the corresponding source lines and guesses at assignment variables at types. Because of the convoluted nature of C programs, we manually checked the results and cleaned up things like macros, typedefs, and parsing errors.

Our model proved mostly correct: less than 1% of pointers are unaligned, and only 1% of integers and 3% of strings point into the heap. About 80% of pointers point to the head of objects. Depressingly, the heap is extremely

Name	Objects	Classes	$p(real\|laika)$	$p(laika\|real)$	$p_{obj}(real\|laika)$	$p_{obj}(laika\|real)$
blackhack	215	6	0.87	1.00	0.87	1.00
xeyes	680	17	0.66	0.68	0.74	0.93
ctorrent	295	19	0.61	0.67	0.60	0.70
privoxy	3881	32	0.90	0.71	0.93	0.82
xclock	2422	54	0.62	0.44	0.72	0.38
xpdf	16846	180	0.61	0.57	0.64	0.56
xarchiver	20993	315	0.52	0.49	0.60	0.60
Average	**6476**	**89**	**0.68**	**0.65**	**0.73**	**0.71**
blackhack-wm	201	8	0.96	1.00	0.96	1.00
ctorrent-wm	249	13	0.80	0.66	0.78	0.73
xeyes-wm	526	22	0.83	0.67	0.79	0.95
privoxy-wm	3615	32	0.92	0.71	0.90	0.88
xclock-wm	2197	43	0.72	0.58	0.79	0.56
xarchiver-wm	7501	89	0.77	0.62	0.80	0.66
xpdf-wm	12995	194	0.63	0.62	0.69	0.64
Average-wm	**3898**	**57**	**0.80**	**0.70**	**0.82**	**0.77**

Table 3.2: Data Structure Detection Accuracy. The first part of the table shows Laika's accuracy using only the memory image; the second part using the memory image and a list of the sizes and locations of objects. $p_{obj}(real|laika)$ and $p_{obj}(laika|real)$ are the accuracy when known atomic types like *int* or *char* are ignored.

---

noisy: on average only 45% of a program's heap address space is occupied by active objects, with the rest being malloc padding and unused or uninitialized chunks. Even more depressingly, only 30% of objects contain a pointer. Since Laika relies on building "pointer fingerprints" to classify objects, this means that the remaining 70% of objects are classed almost entirely by the objects that point to them.

We also encountered a rather disturbing number of poor software engineering practices. Several key X Window data structures, as well as the Perl Compatible Regular Expressions library used by Privoxy, use the dreaded *tail accumulator array*. This archaic programming practice appends a dynamically sized array to a fixed structure; the last element is an array of size 0 and each call to malloc is padded with the size of the array. Although this saves a call to malloc, it makes the software much harder to maintain. This hampers our results on all of the X Window applications, because Laika assumes that all data structures have a fixed length. To express our appreciation, we sent the X Window developers a dirty sock.

We concentrated on the classification accuracy, the chance that Laika actually placed objects from the same real classes together, as opposed to the block accuracy, the chance that Laika generated correct compositions for its classes, because it is more relevant to correctly identifying viruses. There are two metrics: the probability that two objects from the same Laika class came from the same real class, $p(real|laika)$, and the probability that two objects from the same real class were grouped together, $p(laika|real)$. It is easy to see that the first metric could be satisfied by placing all elements in their own classes, while the second could be satisfied by placing all elements in the same class.

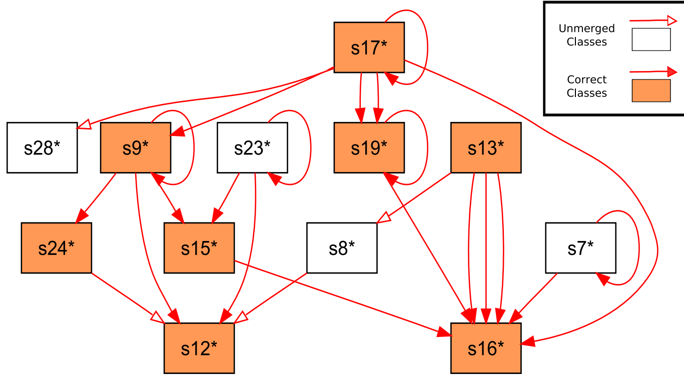


Figure 3.2: Laika generated type graph for Privoxy

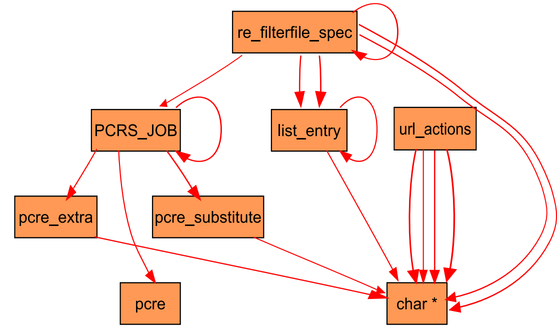


Figure 3.3: Correct type graph for Privoxy

Table 3.2 summarizes the results.

Laika is reasonably accurate but far from perfect, especially on larger programs. The first source of error is data objects (like strings or int arrays), which are difficult to classify without pointers; even some of the real objects contain no pointers. A more interesting problem arises from the variance in size: some classes contain many more objects than others. Generally speaking, Laika merges classes in order to increase the class prior probability $p(\boldsymbol{\omega})$ and splits them in order to increase the the image posterior probability $p(M|\boldsymbol{\omega}, \boldsymbol{X})$. Because the prior has the same weight as a single object, merging two classes that contain many objects will have a much larger effect on the posterior than the prior. For example, Laika may split a binary tree into an internal node class and a leaf node class, because the first would have two address blocks and the second two zero blocks. If there are some 10 objects, then the penalty for creating the additional class would outweigh the bonus for placing the leaf nodes in a more accurate class, but if there are 100 objects Laika is likely to use two classes.

This data also shows just how much Laika's results improve when the random data, freed chunks, and malloc information are removed from the heap. Not only does this result in the removal of 20% of the most random structures, but it also removes malloc padding, which can be uninitialized. Without size information, Laika can easily estimate the size of an object as eight or more times the correct value when there is no pointer to the successor chunk.

To avoid too much mucking around in Laika's dense output files, we can generate relational graphs of the data structures detected. Figure 3.2 shows a graph of the types discovered by Laika for the test application *Privoxy* without the aid of location information; Figure 3.3 shows the correct class relationships. Edges in the graph represent pointers; **s1** will have an edge to **s2** if **s1** contains at least one element of type **s2***. We filtered Figure 3.2 to remove unlikely classes and classes which had no pointers. We are able to easily identify all the correct matching classes (shown shaded) from the given graph, as well as a few classes that were incorrectly merged by Laika (shown in white). For instance, type **s17** corresponds directly to the **re_filterfile_spec** structure, whereas types **s9** and **s23** are, in fact, of the

Bot	μ_{virus}	σ_{virus}	μ_{other}	σ_{other}	Threshold	Samples	Errors	Est. Accuracy	ClamAV
Agobot	0.64	0.038	0.89	0.053	0.75	19/27	0/0	99.4%	83%
Kraken	0.52	0.021	0.83	0.078	0.58	34/27	0/0	99.8%	85%
Storm	0.51	0.005	0.60	0.015	0.53	20/20	0/0	99.9%	100%

Table 3.3: Classification Results. For example, we used 17 of our 34 Kraken samples as training data. When the remaining 16 samples were compared against the signature, they produced an average mixture ratio of 0.52 with a standard deviation of 0.021. Of our 27 clean images, we used 13 as training data, and those produced an average mixture ratio with our Kraken sample of 0.83 with a standard deviation of 0.078. The resulting maximum likelihood classifier classifies anything with a mixture ratio of less than 0.58 as Kraken, and has an estimated accuracy of 99.8%; it classified the remaining 17 Kraken samples and 14 standard Windows applications without error. If a new sample was compared with the Kraken signature and produced a mixture rate of 0.56, it would be classified as Kraken, being 1.9σ from the average of the Kraken samples and 3.5σ from the average of the normal samples.

---

same type and should be merged. In addition, by graphing the class relationship in this manner, visually identifying common data structures and patterns is quite simple. For example, a single self-loop denotes the presence of a singly linked list, as shown by the class **s19/list_entry**.

3.5 Program Detection

Because classifying programs by their data structures already removes all of their code polymorphism, we chose to keep the classifier itself simple. Our virus detector merely runs Laika on the memory images of two programs at the same time and measures how often objects from the different images are placed in the same classes; if many classes contain objects from both programs the programs are likely to be similar. Mathematically, we measured the mixture rate $P(image_i = image_j | class_i = class_j)$ for all object pairs (i, j), which will be closer to 0.5 for similar programs and to 1.0 for dissimilar programs.

Aside from being easy to implement, this approach has several interesting properties. Because Laika is more accurate when given more samples of a class, it is able to discover patterns in the images together that it would miss separately. The mixture ratio also detects changes to the frequency with which the data structures are used. But most importantly, this approach leverages the same object similarity machinery that Laika uses to detect data structures. When Laika makes errors it will tend to make the same errors on the same data, and the mixture ratio focuses on the more numerous - and therefore more accurate - classes, which means that Laika can detect viruses quite accurately even when it does not correctly identify all of their data structures. To focus on the structures that we are more likely to identify correctly, we removed classes that contained no pointers and classes that had very low probability according to the master equation from the mixture ratio. The main disadvantage of the mixture ratio is that the same program can produce different data structures and different ratios when run with different inputs.

We obtained samples of three botnets from Offensive Computing. Agobot/Gaobot is a family of bots based on

GPL source released in 2003. The bot is quite object oriented, and because it is open source there are several thousand variants in the wild today. These variants are often fairly dissimilar, as would-be spam lords select different modules, add code, use different compilers and so on. Agobot also contains some simple polymorphic routines. Storm is well known, and its authors have spent considerable effort making it difficult to detect. Kraken, also known as Bobax, has taken off in the spring of 2008. It is designed to be extremely stealthy and, according to some estimates, is considerably larger than Storm [25]. We ran the bots in QEMU to defeat their VM detection and took snapshots of their memory images using WinDbg.

Our biggest hurdle was getting the bots to activate. All of our viruses required direction from a command and control server before they would launch denial of service attacks or send spam emails. For Agobot this was not a problem, as Agobot allocates plenty of data structures on startup. Our Kraken samples were apparently slightly out of date and spent all their time trying to connect to a set of pseudo-random DNS addresses to ask for instructions; most of the data structures we detected are actually allocated by the Windows networking libraries. Our Storm samples did succeed in making contact with the command servers, but this was not an unmixed blessing as their spam emails brought unwanted attention from our obstreperous network administrators. To this day we curse their names, especially those who are still alive.

These three botnets represent very different challenges for Laika. Agobot is not merely polymorphic; because it is a source toolkit there are many different versions with considerable differences, while Kraken is a single executable where the differences come almost entirely from the polymorphic engine. Agobot is written in a style reminiscent of MFC, with many classes and allocations on the heap. We think Kraken also has a considerable number of data structures, but in the Kraken images we analyzed the vast majority of the objects were from the Windows networking libraries. This means that the Agobot images were dissimilar to each other owing to the many versions, and also to regular Windows programs because of their large numbers of custom data structures, while the Kraken images were practically identical to each other, but also closer to the regular Windows programs. Neither was Laika's ideal target, which would be a heavily object oriented bot modified only by polymorphic routines. Storm was even worse; by infecting a known process its data structures blended with those of services.exe.

Our virus detector works on each botnet separately; a given program is classified as Kraken or not, then Agobot or not, and so on. We used a simple maximum likelihood classifier, with the single parameter being the mixture ratio between the unknown program and a sample of the virus, which acts as the signature. In such a classifier, each class is represented by a probability distribution; we used a Gaussian distribution as the mixture ratio is an average of the individual class mixture ratios. An unknown sample is placed in the most likely class, i.e. the class whose probability distribution has the greatest value for the mixture ratio of that sample. For a Gaussian distribution, this can be thought of as the class that is closest to the sample, where the distance is normalized by standard deviation.

We used half of our samples as training data to estimate the virus and normal distributions. For normal programs we used 27 standard Windows applications including bittorrent, Skype, Notepad, Internet Explorer, and Firefox. To select the signature from the training set, we tried all of them and chose the one with the lowest predicted error rate. Table 3.3 summarizes the results. The detector takes from 3 seconds for small applications up to 20 minutes for large applications like Firefox.

Because Storm injects itself into a known process, we had the opportunity to treat it a little differently. We actually used a clean services.exe process as the "virus"; the decision process is reversed and any sample not close enough to the signature is declared to be Storm. This is considerably superior to using a Storm sample, because our Storm images were much less self-similar than the base services.exe images.

3.5.1 Discussion

The estimated accuracy numbers represent the self-confidence of the model, specifically the overlap of the probability distributions, not its actual tested performance. We included them to give a rough estimate of Laika's performance in lieu of testing several thousand samples. They do not reflect the uncertainty in the estimation of the mean and variance (from some 10-15 samples), which is slightly exacerbated by taking the most discriminatory sample, nor how well the data fit or do not fit our Gaussian model.

It is interesting to note that the accuracy numbers for Kraken and Agobot are roughly comparable despite Agobot containing many unique structures and Kraken using mainly the Windows networking libraries. This occurs because our Kraken samples were extremely similar to one another, allowing Laika to use a very low classification threshold. It is also worth noting that while 99% seems very accurate, a typical computer contains far more than 100 programs.

It would be fairly straightforward to improve our rude 50-line classifier, but even a more complicated version would compare favorably with ClamAV's tens of thousands of lines of code. ClamAV attempts to defeat polymorphic viruses by unpacking and decrypting the hidden executable; this requires a large team of reverse engineers to decipher the various polymorphic methods of thousands of viruses and a corresponding block of code for each.

3.5.2 Analysis

Since our techniques ignore the code polymorphism of current botnets, it is reasonable to ask whether new "memory polymorphic" viruses will surface if data structure detection becomes common. Because virus detection is theoretically undecidable, such malware is always possible, and the best white hats can do is place as many laborious obstacles as possible in the path of their evil counterparts. We believe that hiding data structures is qualitatively more difficult than fooling signature-based detectors, and in this section we will lay out some counter and counter-counter measures to Laika. Our argument runs in two parts: that high-level structure is harder to obfuscate than lower level structure,

and that because high-level structure is so common in programs, we can be very suspicious of any program that lacks it.

Most of the simplest solutions to obfuscating data structures simply eliminate them. For example, if every byte was XORed with a constant, all of the data structures would disappear. While the classifier would have nothing to report, that negative report would itself be quite damning, although admittedly not all obfuscated programs are malware. Even if the objects themselves were obfuscated, perhaps by appending a large amount of random pointers and integers to each, the classifier would find many objects but no classes, which again would be quite suspicious. Slightly more advanced malware might encrypt half of the memory image, while creating fake data structures from a known good program in the other half. Defeating this might require examining the instruction stream and checking for pointer encryption, i.e. what fraction of pointers are used directly without modification.

To truly fool Laika, a data structure polymorphic virus would need to actually change the layout of its data structures as it spreads. It could do this, perhaps, by writing a compiler that shuffled the order of the fields of all the data structures, and then output code with the new offsets. It is obvious that this kind of polymorphism is much more complicated than the kind of simple instruction insertion engines we see today, requiring a larger payload and increasing the chance that the virus would be enervated by its own bugs. The other option would be to fill the memory image with random data structures and hope that the real program goes undetected in the noise. This increases the memory footprint and reduces the stealthiness of the bot, and has no guarantee of success, since the real data structures are still present. Detecting such viruses would probably require a more complicated descendant of Laika with a more intelligent classifier than the simple mixture ratio.

The greatest advantage of Laika as a virus detector is its orthogonality with existing code-based detectors. At worst, it provides valuable defense in depth by posing malware authors different challenges. At best, it can synergize with code analysis; inspecting the instruction stream may reveal whether a program is obfuscating its data structures.

There are two primary disadvantages to using high-level structure. First, a large class of malware has no high-level structure by the virtue of simplicity. A small kernel rootkit that overwrites a system call table or a bit of shellcode that executes primarily on the stack won't use enough data structures to be detected, and distinguishing a small piece of malware inside a large program like Apache or Linux is more difficult than detecting it in a separate process. However, we believe that there is an important difference between fun and profit. Turning zombie machines into hard currency means putting them to some purpose, be it DoS attacks, spam, serving ads, or other devilry, and that means running some sort of moderately complex process on the infected machines. It is this sort of malware that has been more common lately [1, 22], and it is this sort of malware that we aim to detect.

The second main disadvantage is the substantial increase in resources, in both memory and processor cycles, when compared to current virus scanners. Although a commercial implementation would no doubt be more highly

optimized, we do not believe that order of magnitude improvements are likely given the computational complexity of the equations to be solved. Moreover, our data structure detection algorithms apply only to running processes. Worse, those processes will not exercise a reasonable set of their data structures unless they are allowed to perform their malicious actions, so a complete solution must provide a way to blunt any malicious effects prior to detection. This requires either speculatively executing all programs and only committing output after a data structure inspection, or recording all output and rolling back malicious activity. Although we believe that Moore's law will continue to provide more resources, these operations are still not cheap.

3.5.3 Scaling

Laika runs in linear time in the number of viruses, with moderately high constants. Clearly a commerical version of Laika would require some heuristical shortcuts to avoid stubbornly comparing a test image against some 10^5 virus signature images. The straightforward approach would be to run Laika on the unknown program, record the data structures, compute a signature based on those data structures, use that signature to look up a small set of similar programs in a database, and verify the results with the mixture ratio.

It turns out this is not completely straightforward after all. Consider the most natural approach. The list of data structures may be canonicalized by assigning each structure a unique id. Typed pointer issues can be avoided by assigning a structure a pointer only after canonicalizing the structures to which it points. A program image could then be represented as a vector of structure counts and confidences, and the full mixture ratio computed only for the k nearest neighbors under some distance metric. This approach is extremely brittle. Imagine we have a data structure where one field is changed from 'pointer' to 'zero'. This not only changes the id of that data structure, but also all structures that point to it. In addition, the vector would lie in the space of all known data structures, which would make it hundreds of thousands of elements long. We believe that these difficulties could be solved by a bit of clever engineering, but we have not actually tried to do so.

3.6 Related Work

Most of the work on the semantic gap so far has come from the security community, which is interested in detecting viruses and determining program behavior by "looking up" from the operating system or VMM towards the actual applications. While most of these problems are either extremely computationally difficult or undecidable in theory, there are techniques that work in practice.

3.6.1 The Unobfuscated Semantic Gap

Recently, many researchers have proposed running operating system services in separate virtual machines to increase security and reliability. These separated services must now confront the semantic gap between the raw bytes they see and the high-level primitives of the guest OS, and most exploit the fixed interfaces of the processor and operating system to obtain a higher level view, a technique known as virtual machine introspection. Antfarm [32] monitors the page directory register (CR3 in x86) to infer process switches and counts, while Geiger [33] monitors the page table and through it can make inferences about the guest OS's buffer cache. Intrusion detectors benefit greatly from the protective isolation of a virtual machine [10, 23]; most rely on prior information on some combination of the OS data structures, filesystem format, and processor features. Polishchuk *et al.* [53] also attempt to determine heap types, but they do so from a supervised environment, where exact malloc locations and debug information are available, making the problem considerably easier.

3.6.2 The Obfuscated Semantic Gap

Randomization and obfuscation have been used by both attackers and defenders to make bridging the semantic gap more difficult. Address space randomization [5], now implemented in the Linux kernel, randomizes the location of functions and variables; buffer overflows no longer have deterministic side effects and usually cause the program to crash rather than exhibit the desired malicious behavior.

On the other side, virus writers attempt to obfuscate their programs to make them more difficult to disassemble [40]. Compiler-like code transformations such as nop or jump insertion, reordering, or instruction substitution [35] are relatively straightforward to implement and theoretically extremely effective: universal virus classification is undecidable [14] and even detecting whether a program is an obfuscated instance of a polymorphic virus is NP-Complete [11]. Even when the virus writer does not explicitly attempt to obfuscate the program, new versions of existing viruses may prove effectively polymorphic [26].

3.6.3 The Deobfuscated Semantic Gap

Polymorphic virus detectors usually fall into two classes: code detectors and behavior detectors. For example, Christodorescu *et. al* [13] attempt to detect polymorphic viruses by defining patterns of instructions found in a polymorphic worm and searching for them in the unknown binary. Unlike a simple signature checker, these patterns are fairly general and their virus scanner uses a combination of nop libraries, theorem proving and randomized execution. In the end it is capable of detecting instruction (but not memory) reordering, register renaming, and garbage insertion. A recent survey by Singh and Lakhotia [57] gives a good summary of this type of classifier. In addition, Ma *et. al* [43]

attempt to classify families of code injection exploits. They use emulated execution to decode shellcode samples and cluster results on executed instruction edit distance. Their results show success in classifying small shellcode samples, but they rely entirely on prior knowledge of specific vulnerabilities to locate and extract these samples.

The second class of detectors concentrate on behavior rather than fingerprinting. These methods usually have either a group of heuristics for malicious behavior [39] or statistical thresholds [20]. Often they concentrate on semantically higher level features, like system calls or registry accesses rather than individual instructions. Because they are not specific to any particular virus, they can detect unknown viruses, but they often suffer from false positives since benign executables can be very similar to malicious ones.

3.6.4 Shape Analysis

Modern compilers spend a great deal of time trying to untangle the complicated web of pointers in C programs [61]. Many optimizations cannot be performed if two different pointers in fact refer to the same data structure, and answering this question for structures like trees or hash tables can be difficult. Shape analysis [16, 24] attempts to determine the high-level structure of a program: does a set of allocations form a list, binary tree, or other abstract data type? Although it can enable greater parallelism, shape analysis is very expensive on all but the most trivial programs. Our work attacks the problem of high-level structure from a different angle; although we do not have the source code of the target program, our task is simplified by considering only one memory image.

3.6.5 Reverse Engineering

Reverse engineering is the art of acquiring human meaning from system implementation. However, most of the work in this field is concentrated on building tools to aid humans discover structure from systems [31, 62], rather than using the information directly. Furthermore, a large amount of the reverse engineering literature [49, 66] is concerned with reverse engineering structure from source code to provide developers with high-level understanding of large software projects.

A more superficially similar technique is Value Set Analysis [3]. VSA can be thought of as pointer aliasing analysis for binary code; it tries to determine possible values for registers and memory at various points in the computation. It is especially useful for analyzing malware. Laika differs from VSA in that it is dynamic rather than static, and that Laika's output is directly used to identify viruses rather than aid reverse engineers.

3.7 Conclusions

In this paper we have discussed the design and implementation of Laika, a system that detects the data structures of a process given a memory image. The data structures generated by Laika proved surprisingly effective for virus detection. The vast majority of current polymorphic virus detectors work by generating low level fingerprints, but these fingerprints are easily obfuscated by malware writers. By moving the fingerprint to a higher level of abstraction, we increase the difficulty of obfuscation. Laika also provides valuable synergy to existing code signature based detectors.

Laika exploits the common humanity of programmers; even very flexible fixed interfaces like Von Neumann machines are often used in standard ways. Because fixed interfaces dominate the landscape in systems - often the interface *is* the system - there should be many other such opportunities.

Chapter 4

Macho: Programming with Man Pages

4.1 Introduction

Programming is hard. Because computers can only execute simple instructions, the programmer must spell out the application's behavior in excruciating detail. Because computers slavishly follow their instructions, any trivial error will result in a crash or, worse, a security exploit. Together they make computer code difficult and time consuming to write, read, and debug.

Programmers write software the same way they do everything else: by imitating other people. The first response to a new problem is often to google it, and ideally find code snippets or examples of library calls. The programmer then combines these chunks of code, writes some test cases, and makes small changes to the program until its output is correct for the inputs he has considered.

Software engineering researchers have developed techniques to help automate each of these parts of the programming process. Code search tools scan through databases of source code to find code samples related to programmer queries. For example, SNIFF [9] uses source code comments to help find snippets of code, and Prospector [45] finds library calls that convert from one language type to another. Automated debugging tools not only help find problems [68], but sometimes even suggest solutions [69]. For example, recent work by Weimer *et al.* [64], describes how to use genetic programming algorithms to modify buggy source code automatically until the modified programs pass a set of test cases.

Although these techniques do save time, the programmer is still responsible for selecting code snippets, arranging them into a program, and debugging the result. In this paper we describe Macho, a system that generates simple Java programs from a combination of natural language, examples (unit tests), and a large repository of Java source code (mostly from Sourceforge projects). It contains four subsystems: a natural language parser that maps English into database queries, a large database that maps programmer abstractions to snippets of Java code, a stitcher that combines code snippets in "reasonable" ways, and an automated debugger that tests the resulting candidate programs against the examples and makes simple fixes automatically.

Because database search and automated debugging are still hard problems with immature tools, Macho's abilities

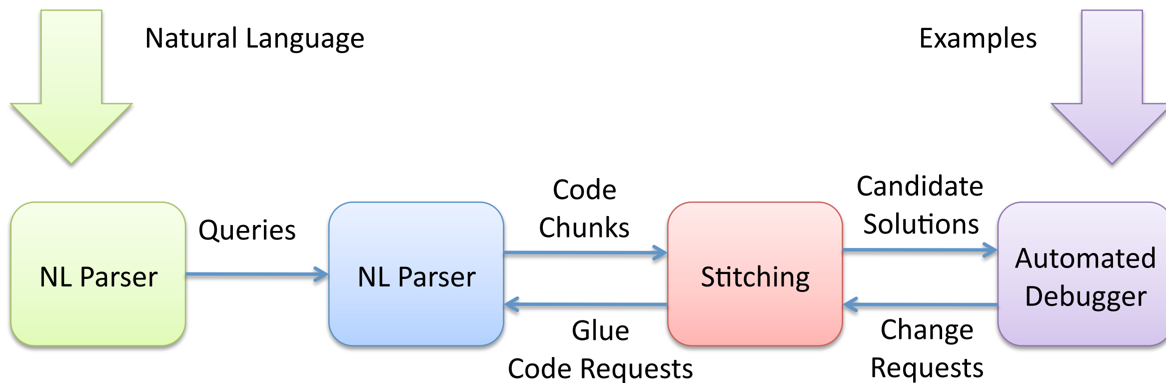


Figure 4.1: Macho workflow

are correspondingly basic. Our current version of Macho was able to synthesize simple versions (no options, one or two arguments) of various Unix core utilities from simple natural language specifications and examples of correct behavior, including versions of ls, pwd, cat, cp, sort, and grep. Macho was unable to generate correct solutions for wget, head, and uniq.

Macho is a remarkably simple attack on an extraordinarily difficult task. Natural language understanding is considered one of the hardest problems in Artificial Intelligence with a huge body of current research. Generalizing from examples is similarly difficult. And even once a computer system "understands" the problem it still must actually write suitable Java code.

Our key insight is that natural language and examples have considerable synergy. Macho has a fighting chance to generate correct programs because each component can partially correct for the mistakes of the others. For example, a database query will return many possible results, most of which will be incorrect, but by leveraging the type system the stitcher can eliminate many unlikely solutions. Even more importantly, the test cases allow Macho to partially detour around the difficult problem of natural language processing. Modern machine learning techniques provide probabilistic answers, whether the question is the meaning of a piece of natural language or the best sample function in the database to use. Backed by its automated debugger, Macho can afford to try multiple solutions.

In addition, combining examples and natural language greatly reduces their ambiguity: the set of programs that satisfies both the natural language and the test cases is much smaller than the sets that satisfy each input individually. Just passing one or two test cases is no guarantee the program will correctly handle others. For example, Macho found it surprisingly easy to synthesize cat from a unit test using the empty files it used for generating ls. However, we found that most of the time a program that passed even one reasonable test case would be correct. Together natural language and examples form a fairly concrete specification.

4.2 Design and Implementation

Macho's workflow mirrors a human programmer. It maps the natural language to implied computation, maps those abstractions to concrete Java code, combines the code chunks into a candidate solution, and finally debugs the resulting program. All of these tasks are open problems for which we have only rudimentary solutions. The key advantage of the unit test is that Macho can use modern probabilistic machine learning techniques to generate multiple solutions and select the one(s) that pass the test. However, this results in a rapidly increasing number of candidates as each stage passes multiple results on to the next. The goal of each subsystem is therefore to minimize the amount of brute force and thereby synthesize the largest possible programs. Our evaluation contains an example of natural language, database, stitching, and final output for a sample program in figure 4.6.

4.2.1 Natural Language Parser

> A woman asks her husband, a programmer, to go shopping: "Dear, please, go to the grocery store to buy some bread. Also, if they have eggs, buy 6."
>
> "O.K., honey."
>
> Twenty minutes later the husband comes back bringing 6 loaves of bread. His wife is flabbergasted, "Dear, why on earth did you buy 6 loaves of bread?"
>
> "They had eggs."
>
> *–How to tell if someone is a Programmer*

Our natural language parsing subsystem attempts to extract implied chunks of computation and the data flow between them from the words and phrases it receives, and encode that knowledge for the database. Usually the structure of the sentence can be directly transformed to requested computation: verbs imply action, nouns imply objects, and two nouns linked by a preposition imply some sort of conversion code. This mapping is conceptually similar to previous work [6], but Macho's database "understands" a much larger number of concepts, including abbreviations. In order to handle these more varied sentences, we began with an off-the-shelf system provided by the University of Illinois Cognitive Computation group to tag individual words with their part of speech (noun, verb, adjective, etc.) and to split sentences apart into smaller phrases. Chapter 2 contains a simple explanation of these parsers.

Our main problem was fixing the errors of the parser, which was trained on a standard corpus of newspaper articles, not jargon filled man pages. For example, 'file' is usually a verb, like "the SEC filed charges against Enron today." and print is often a noun, e.g., "Their foul prints will not soon be cleansed from the financial system.". These kinds of errors were quite common.

```
public static void main(String[] args) {

   //first (original) database
   //function call matching 'directory -> files'
   files = directory.listFiles();

   //third (final) database
   //expression (multiple operations) matching 'directory -> files'
   files = new File(directory).listFiles();

   //third (final) database
   //expression with static dataflow matching 'directory -> files' through variable tmp
   //generates same pattern as the previous example.  The if statement
   //would have been caught by the phantom second database.
   tmp = new File(directory);
   if(foo)
      files = tmp.listFiles();

   //typical useless expression: method with some useless syntax
   tmp = new File(directory + "/blah").listFiles();

   //We made no effort to canonicalize the expressions
   a = b + (c + d) != (b + c) + d;
}
```

Figure 4.2: Examples of patterns mined by the various databases. In each case, a pattern is a block of code (a single function in the first database, a full Java expression in the third) which matches the names expressed in a query. All of the functions in this figure match the pattern 'has an input variable named directory and an output variable named files'.

To help detect what words were intended to act as actions, we build a graph of prepositions linking the objects in a sentence together into a tree. A traversal of this tree reveals the relationship between the nouns at its leaves. When we find words that are not linked to the rest of the sentence by this graph, we can guess that they are misclassified verbs. The parser also provides some hints as to likely control flow. For example, plural adjective or adverbial phrases often imply a filter operation that is implemented as an if statement. The description of grep contains ‘lines matching a pattern’ which implies only some lines will be used.

By looking at the language constructs that connect various nouns, the parser can determine what nouns are likely the same object. When a person says “text file” they do not mean a text and a file, but a single object. If those nouns were linked by a preposition (“text in a file”), then they would be interpreted differently. These relationships are passed along to the database and help narrow down which frequent expressions might relate to our program.

Since our parser’s goal was to understand likely relationships between words and phrases, we did not worry too much about perfectly understanding the grammar. Imperfect grammar in input sentences (such as the language often used in man pages) does not noticeably decrease the effectiveness of our parser.

4.2.2 Database

As the subsystem that maps natural language abstractions to concrete Java code, the database is the engine that powers Macho. When the database can suggest reasonable code chunks, the stitching can usually find a correct solution, but when the database fails the space of candidate programs is simply too large to succeed by flailing randomly.

Our original plan was to use Google Code, but we almost immediately dismissed it as completely inadequate. Google Code indexes a huge number of files, but it appears to only perform keyword search on the raw text of the source files, which we found to be inadequate for our problem. Instead, we developed our own database for Macho.

Our first step was to obtain a data set of about 200,000 Java files from open source projects and compile them using a special version of javac that we modified to emit abstract syntax trees. We compiled rather than parsed because we wanted exact global locations for each function call, and because we didn't want to reuse broken code. Since open source programmers are not exactly paragons of code maintenance, only about half of our source files compiled successfully.

Our first database returned candidate methods based on input and output variables, e.g. the query $directory \rightarrow files$ would return all functions called with an input variable named directory and assigned to a variable named files. This nicely captured the different abstractions that different programmers used to represent code, which is important because functions have only one name. The problem with this approach is that many things aren't usually implemented as functions. Higher level concepts like ignore, first, or adjacent usually appear as operations or even control flow. Often they have no input variables or are only tagged in the comments.

Our second database returned raw frequent patterns from the code graph. We started with an abstract syntax tree for each source code file and connected these trees by adding edges for cross-file relations, like linking method calls with their definitions. We then mined the graphs looking for frequent subgraphs that we planned to use as code snippets. In theory, this graph represents all of the information in the data set in a concise, uniform fashion. In practice, the frequent subgraph set is impossibly large. Every frequent graph of size n can contain up to 2^n subgraphs, all of which are also frequent. The tool we used [67] generated thousands of patterns for even a few files.

Our third and final database returns raw frequent expressions, i.e. combinations of operations and functions without control flow. Figure 4.2 shows examples of patterns detected by each of the databases. This cuts out a huge number of edges in the graph, including all control flow. Expressions are also trees, not graphs, which makes them much easier to mine without the problem of subgraph isomorphism. Since we made no effort to canonicalize the expressions, our miner simply counted occurrences of expression trees (and their subtrees) by straightforward recursive equality testing, i.e. two trees are identical if their root nodes and subtrees are identical.

These cuts dramatically reduce the number of frequent patterns (we mined 400,000 expressions, or about 4 per file) even after we used static dataflow to generate expressions across variable definitions. However, we found that most of

the extended patterns were either very infrequent relative to the function calls they were composed of, or consisted of a standard operation like plus or array access wrapped around a method. We were also disappointed by the comments. It seems that most of the keywords are dominated by basic operations like increment that are used frequently.

Because expressions can be used exactly like functions in Java, the third database is indexed identically to the first, i.e. the query $file \rightarrow lines$ returns expressions with an input variable named file and an output variable named $lines$. In the end, however, we mostly restricted the expressions to single functions - the same results we would get from the first database. The full expressions were primarily useful for Prospector [45] style type conversions to glue code from other database queries together.

If an expression was seen n times with the requested keyword and m times total, the score for that expression will be n^2/m, which attempts to strike a balance between fit and frequency. Because it is not always clear whether a plural makes sense, every variable query is computed twice, once normally and once after applying the Porter stemming algorithm. The results are merged with equal scores. Currently the input and output scores are computed separately, which means that the sets of expressions that generated the two queries may be disjoint.

4.2.3 Stitching

Macho's stitching subsystem combines results from database queries into candidate programs. Its main guide is the type system; two expressions can be linked by a variable if the output type of one matches the input type of the other. If the types don't match, the stitcher will query the database for common chunks of code that were used to convert between those types.

Macho also generates a small amount of control flow. If statements are generated only from hints by the natural language parser and the synthesizer. Map loops are generated when suggested by the type system. Macho tries to limit control flow generation because it swiftly increases the solution space; an upstream chunk may be placed in any block above the downstream chunk.

The most difficult part of stitching is keeping track of the data flow between expressions in the presence of control flow. The natural language gives a great deal of information for how information is supposed to flow from one chunk to another; previous natural language programming systems generated code without any search at all. This is actually kind of an interesting problem: current programming languages are designed (at least in theory) to be easy for humans, not machines, to deal with, and compiler transformations are performed only after converting the code to an intermediate representation. However, we can't just switch to a Macho-oriented language, whatever that would be, because then we would have no training data.

4.2.4 Automated Debugger

Macho's automated debugging subsystem attempts to debug candidate programs. This type of automated debugging is potentially extremely difficult, but many of the automatically generated candidate programs will have utterly obvious errors that can be fixed easily. The primary difference between stitching and automated debugging is that debugging is dynamic rather than static and has access to the behavior of the program. Currently the automated debugger runs the candidate in a sandbox and performs a diff between the output of the candidate and the unit test and classifies the candidate into one of five simple cases:

- No output because an exception was thrown. The automated debugger will then request the stitching tool to insert an if block surrounding the offending statement. The stitching will then try common tests (boolean functions with no side effects) based on variables in scope for the conditional.
- Correct output. The solution is returned to the user.
- Subset of the correct output. In this case the automated debugger will insert print statements for all of its variables and pray.
- Superset of the correct output. In this case the automated debugger tries to identify which print statement is generating the additional output and again request an if statement surrounding the questionable print statement.
- Garbage. The output is completely incorrect, and the solution is junked.

These components have synergy beyond simply correcting mistakes. For example, our automated debugger leverages the database to suggest changes to buggy programs. When it is faced with a potential solution for ls which incorrectly prints hidden files, the debugger queries the database for commonly used functions of *java.io.File* which could be used in an if statement to restrict the obstreperous print. This simple probabilistic model allows it to try the *isHidden* method even though it is not used elsewhere in the candidate solution.

Although the automated debugging seems superficially simple, it actually solves a very difficult problem of library combination. Macho's database finds candidate functions entirely be name, which may be unrelated to their purpose. Running the code allows the debugger to eliminate these imposter functions.

4.3 Evaluation

Objectively evaluating Macho is very difficult. There is no standard test suite where we can benchmark our results against other systems, and using the language from the man pages directly is almost impossible. Consider the byzantine man page description for wget:

Program	Result	Input	Notes
pwd	success	Print the current working directory.	Difficult as there is no input.
pwd	success	Print the user directory.	CWD = “user.dir” in Java.
pwd	success	Print the current directory.	Abbreviation!
pwd	fail	Print the working directory.	Breaks NLP for arcane reasons.
pwd	fail	Show the current working directory.	Database entries for show are mostly graphics.
cat	success	Print the lines of a file.	Vanilla.
cat	success	Read a file.	Print is synthesized.
cat	fail	Display the contents of a file.	Database entries for contents are mostly graphics.
cat	fail	Print a file	Solutions print the file name.
sort	success	Sort the lines of a file.	Print is synthesized.
sort	success	Sort a file by line.	
sort	fail	Sort a file.	Insufficiently precise specification.
sort	fail	Sort the contents of a file	We didn’t add contents = ReadAllLines().
grep	success	Print the lines in a file matching a pattern.	Solutions using both JavaLib and GNU regexes.
grep	fail	Find a pattern in the lines of a file.	Correct except for if statement linking test and print.
grep	fail	Search file for a pattern.	Poor resiliency for function names.
ls	success	Print the names of files in a directory. Sort the names.	
ls	success	Print the contents of a folder. Sort the names.	
ls	fail	Print the names of the entries in a directory.	Entries to names fails.
ls	fail	Print the files in a directory.	Does not synthesize sort.
cp	success	Copy src file to dest file.	Programmer abbreviation!
cp	success	Copy file to file.	Ugly but Macho needs to know there are two inputs.
cp	fail	Duplicate file to file.	No candidate in database.
wget	fail	Download file.	Candidates have extra functionality.
wget	fail	Open network connection. Download file.	Macho can’t create buffer transfer loop.
head	fail	Print the first ten lines of a file.	’First’ is incomprehensible.
uniq	fail	Print the lines of a file. Ignore adjacent lines.	’Ignore’ and ’adjacent’ don’t map to libraries.
perl	fail	The answer to life, the universe, and everything.	Seems to work, but it’s still running.

Figure 4.3: Macho’s results for generating select core utils. This figure shows the results for pwd, cat, sort, grep, ls, cp, wget, head, and uniq, the natural language input we used for each of these programs, and a very short description of why Macho succeeded or failed. We discuss each example in detail in the text.

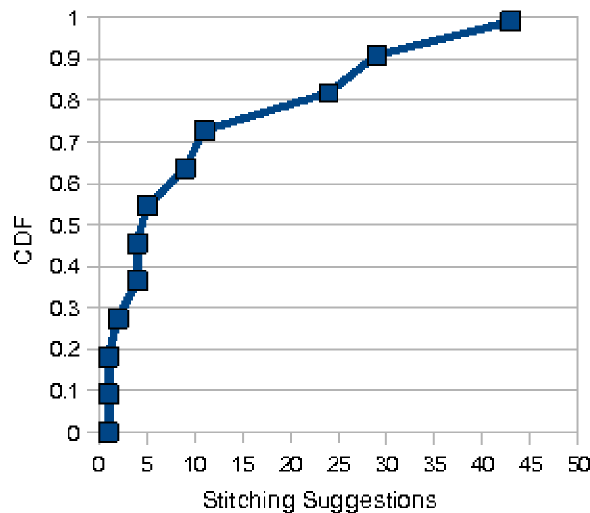


Figure 4.4: This figure shows the number of stitching candidates that the automated debugger had to examine before it could find one it could patch up to pass the unit test. It shows that Macho is not just flailing randomly - many programs are solved with relatively few stitching candidates. Interestingly the most difficult program to synthesize was actually pwd.

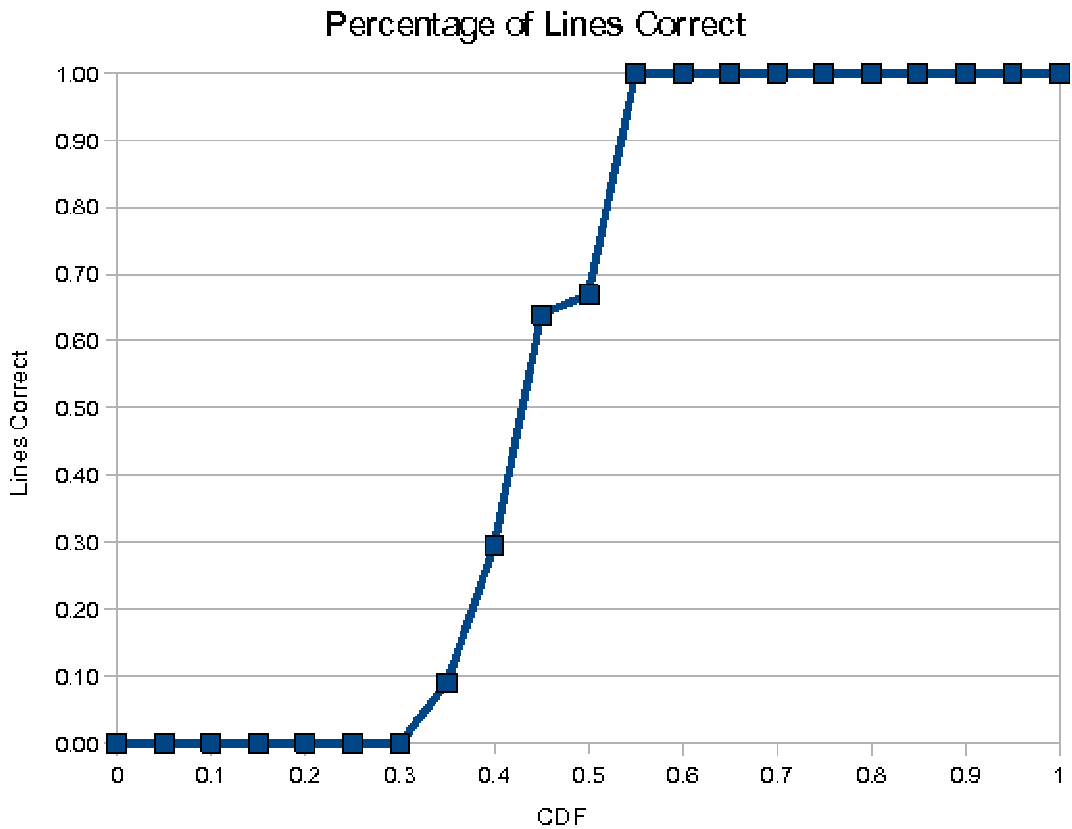


Figure 4.5: This figure shows the percentage of lines of code correct of Macho's *best* solution after stitching and synthesis (the average number of lines correct for a stitching solution would be considerably lower) as determined by me. For the most part Macho either succeeds beautifully or fails horribly, which explains the sharp phase transition. I selected lines of code rather than comparing the outputs because even a few lines of erroneous code are usually enough to completely alter the output. For details on which programs were partially successful, see the appendix.

> GNU Wget is a free utility for non-interactive download of files from the Web. It supports HTTP, HTTPS, and FTP protocols, as well as retrieval through HTTP proxies.

Giving out partial credit is also difficult. Some of Macho's solutions are very close but not byte identical, but automatically determining whether or not an output is sufficiently close to the test case is approximately as hard as generating the program, an artificial version of the Dunning-Kruger effect. Under these circumstances we decided to try to pick an interesting set of natural language inputs right on the border of Macho's capabilities and use our best judgement when the test cases were "close".

Macho succeeded in generating simple versions of six out of nine coreutils - pwd, cat, sort, grep, cp, and ls - and failed to synthesize wget, head, and uniq. For each core utility, we targeted its default behavior: no options and the

minimum number of arguments possible. Since we had the programs available anyway, we used them to generate our unit tests. All of the programs had only one short test and the results are shown in Figure 4.3.

4.3.1 pwd

Macho solved pwd fairly easily. There are two ways to get the current working directory in Java, `new File(``.'')` and `System.getProperty(``user.dir'')`, but the first prints an additional "/.". When a candidate solution has unfilled inputs but no additional queries, Macho looks up common literals for that expression in the database. The database was able to find `System.getProperty()` for "current working directory", "current directory", "working directory", and "user directory", but "show" only returned GUI functions like `JFileChooser.ShowOpenDialog()`. "Print the working directory" confuses the natural language parser because it thinks print is a noun and our usual heuristics for fixing its output do not work in this case.

4.3.2 cat

Cat was surprisingly tricky. The query $file \rightarrow lines$, a very common pattern in the core utilities, almost never occurs in real Java, because reading in an entire file at once wastes a great deal of memory, so we wrote `ReadAllLines()` ourselves. We were also forced to manually increase the score of `System.out.println()`, because it is usually called with a string concatenation of variables and formatting constants. Even without the special sauce, though, Macho comes very close to solving cat: its solution constructs the necessary `BufferedReader` and `FileReader`, but does not try a loop around `readLine()`. In our database, 1471 out of the 2130 calls of `readLine()` are enclosed in a loop, so it should not be too hard to have Macho automatically try loops for such functions.

Although we gave ourselves credit for cat, Macho's solution is sometimes one byte off. GNU cat prints the exact bytes of the file. Macho's version prints every line in the file followed by a linefeed. This will sometimes result in an extra newline.

Macho fails on "Display the contents of a file" because, like "show", "display" is usually used for GUI functions. "Print a file" simply prints the name of the file, which is mildly reasonable but wrong.

4.3.3 sort

Sort is basically cat with sorted output, and Macho succeeded in constructing it from both "Sort the lines of a file" and "Sort a file by line". Interestingly, coreutils sort and ls sort their items in a different order on Gentoo and Fedora Core Linux. We hoped Macho would return different implementations for these cases depending on which OS was used, but it turns out that none of the sorting functions returned by the database are configurable. `Arrays.sort()` and Gentoo Linux sort in ASCII order; Fedora Core mixes upper and lower case, i.e. "A B a b" vs "A a B b". We think this

is actually an excellent example of the usually meaningless but occasionally critical details that make programming so much fun.

Macho completely fails on “Sort a file”, which is after all quite ambiguous, and also on “Sort the contents of a file”, since our `ReadAllLines()` function was never called with an output variable called contents.

4.3.4 cp

Cp was deceptively tricky. Although there were many implementations of file copy in our database, most of them - including our favourite one, from the popular online web game *Kingdom of Loathing* - contained additional log statements, showing the danger of randomly reusing suitably appearing libraries without checking their implementations. Fortunately Macho's unit test was able to weed out the offenders. Cp also ran afoul of programmer conciseness. Programmers abbreviate by default, so $copy(src, dest)$ actually works better that $copy(source, destination)$ which evidently was too verbose for most programmers. This also means that $dir \rightarrow files$ works just as well as $directory \rightarrow files$.

The database failed on “Duplicate file” since the word copy is so ubiquitous in comparison.

4.3.5 wget

Wget was similar to cp, but Macho was unable to find a virgin implementation of download file. What appears to be the excellent candidate `downloadFile(<File>, <URLConnection>)` is actually from a Java version of the Debian Linux package tool *apt*, and not only downloads a file but also fetches and applies an MD5 checksum. We also tried the more descripte “Open network connection. Download file” which comes much closer. Macho can generate the open connection and, like cat, the set of readers and calls required, but does not put a loop around `readLine()`. However, since wget is frequently called with binary files (somewhat more frequently than cat) this is only a partial solution. Macho does find the function to read a byte array from from the connection, but cannot create an appropriately sized array.

4.3.6 grep

Grep was trickier because the natural language specification involved control flow. Macho creates an if statement from the natural language hint to only print matching lines. The primary challenge was choosing a function to decide whether or not a line matched. Macho's first choice was `startsWith()`, but it also found slightly more complicated solutions using both the Java and GNU regular expression libraries. Macho almost succeeded with “Find a pattern in the lines of a file.”, where it generates all the correct loops and function calls but does not try adding an if statement

connecting the regular expression search to the print. “Search file for a pattern” fails because Macho cannot find an appropriate function named search.

4.3.7 ls

Ls required the most substantial synthesis, because the natural language specification does not mention that ls should not print hidden files, or that if it is called with a single file as its argument it should print the filename, because that kind of thing is tedious to describe verbally. Synthesis for ls generates two if statements, one guarding `Arrays.sort()` which throws an exception when called with the null array returned by `java.io.listFiles()` returns when called with a file instead of a directory, and the final print statement, in case the file begins with a period. It also generates an additional print statement which is necessary when ls is called with a single file. Figure 4.7 shows the final solution for ls after three rounds of feedback between the synthesis and stitching stages.

Macho was also able to succeed with “Print the contents of a folder. Sort the names” where it looks up the file to string conversion, but it failed for “Print the names of the entries in a directory.” where it tried functions like `java.util.Map.Entry.getKey()` and for “Print the files in a directory.” where the debugger did not attempt to sort the results.

Ls would be much easier to synthesize if it printed hidden files. There are two functions to list the files in a directory, `java.io.File.listFiles()`, which returns an array of files, and `java.io.File.list()`, returns an array of file names. Because Macho needs a file object to test whether or not the file is hidden, solutions using the second version (which appeared quite frequently) will not work.

4.3.8 Head and Uniq

Head and uniq (unsolved) are difficult because they use more abstract natural language: the database results for queries like $lines \rightarrow first$ or $lines \rightarrow adjacent$ are totally useless. Part of the problem is that Macho is not intelligent enough to notice that lines is a sequence. The database had other interesting problems as well. Because Java has excellent cross-platform GUI support, it is not usually the first choice for the kind of file operation intensive command-line applications we built. For example, the variable “lines” is represented by an integer more than a string, as Eclipse uses line numbers frequently.

Natural Language: Print the names of files in a directory. Sort the names.
Queries: $directory \rightarrow files$, $files \rightarrow names$, $print(names)$, $sort(names)$

Query	Score	Java Expression
$files \rightarrow names$	0.28	<UNSET File>.getName()
$files \rightarrow names$	0.10	<UNSET org.eclipse.swt.widgets.FileDialog>.open()
$files \rightarrow names$	0.09	<UNSET org.armedbear.j.File>.getName()
$files \rightarrow names$	0.08	<UNSET File>.toString()
$directory \rightarrow files$	0.15	<UNSET File>.listFiles()
$directory \rightarrow files$	0.14	new File(<UNSET File>, <UNSET String
$directory \rightarrow files$	0.09	new File(<UNSET String>)
$directory \rightarrow files$	0.09	<UNSET File>.listFiles(<UNSET DirectoryFilter>)
$java.String \rightarrow java.File$	0.45	new File(<UNSET String>)
$java.String \rightarrow java.File$	0.25	new File(<UNSET File>, <UNSET String>)
$java.String \rightarrow java.File$	0.14	new File(<UNSET String>, <UNSET String>)
$java.String \rightarrow java.File$	0.04	File.createTempFile(<UNSET String>, ...)
$java.File \rightarrow boolean$	0.26	<UNSET File>.exists()
$java.File \rightarrow boolean$	0.15	! <UNSET File>.exists()
$java.File \rightarrow boolean$	0.14	<UNSET File>.isDirectory()
$java.File \rightarrow boolean$	0.10	<UNSET File> != null

```
//First solution for ls, score 0.0021
public static void Ls1(java.lang.String p_directory) {
  java.io.File tmp = new File(p_directory);
  java.io.File[] files = tmp.listFiles();
  int tmp_0 = files.length;
  java.lang.String[] tmp_1 = new java.lang.String[tmp_0];
  for(int tmp_3 = 0; tmp_3 < files.length; ++tmp_3)  {
    java.io.File tmp_2 = files[tmp_3];
    java.lang.String names = tmp_2.getName();
    tmp_1[tmp_3] = names;
  }
  Arrays.sort(tmp_1); //638
  for(int tmp_5 = 0; tmp_5 < tmp_1.length; ++tmp_5)  {
    java.lang.String tmp_4 = tmp_1[tmp_5];
    System.out.println(tmp_4);
  }
} //needs synthesis, but no place to attach isHidden() in the print loop

//Fifth solution for ls, score 0.0015
public static void Ls2(java.lang.String p_directory) {
  java.io.File tmp = new File(p_directory)
  java.io.File[] files = tmp.listFiles();
  Arrays.sort(files);
  for(int tmp_1 = 0; tmp_1 < files.length; ++tmp_1)  {
    java.io.File tmp_0 = files[tmp_1];
    java.lang.String names = tmp_0.getName();
    System.out.println(names);
  }
} //needs synthesis
```

Figure 4.6: Final solution for ls.

```
public static void Ls3(String p_dir) {
  java.io.File tmp = new File(p_dir);
  java.io.File[] files = tmp.listFiles();
  boolean tmp_3 = tmp.isDirectory();
  if(tmp_3) {
    Arrays.sort(files);
    for(int tmp_1 = 0; tmp_1 < files.length; ++tmp_1)  {
      java.io.File tmp_0 = files[tmp_1];
      java.lang.String names = tmp_0.getName();
      boolean tmp_2 = tmp_0.isHidden();
      if(tmp_2) {
      }  else  {
        System.out.println(names)
      }
    }
  }  else  {
    System.out.println(tmp + "");
  }
} //correct
```

Figure 4.7: Final solution for ls.

4.4 Lessons Learned

4.4.1 The Database is King

Although most of the programs Macho writes are 10-15 lines or less, there are a *lot* of potential 10-line Java programs. Brute force really does not get very far - the ability of the database to select reasonable pieces from the natural language heuristics is absolutely critical. In general, when the stitching failed, it was often reasonable to think of a hack, or a simple fix, or just let it run a little longer, but when the database failed Macho had no hope of ever generating a correct or even partially correct solution. Improving Macho will require a superior database above everything else.

4.4.2 Pure NLP is Bad

Programming with natural language is generally considered a bad idea because specifying details gradually mutates the natural language into a word versions of Visual Basic. Consider a natural language spec for ls:

```
Take the path "/home/zerocool/" If the path is a file, print it.  Otherwise get
the list of files in the directory.  Sort the result alphabetically.  Go over the
result from the beginning to the end:  If the current element's filename does not
begin with ".", print it.
```

which is our best guess for the input required for Pegasus [37]; it is obvious why most programmers would prefer to use Python instead. Instead, a Macho programmer can specify the basic task very simply:

```
Print the names of files in a directory.  Sort the names.
```

Even an almost trivial program like this leaves many details unspecified: should the sort be alphabetically by filename, size, file extension, or date? Should the program print the full path, the relative path, or just the name of the files? Does “files” include subdirectories or hidden files? All of these questions are easily cleared up by an example of correct operation. Such examples not only have a higher information density than tedious pages of pseudocode or UML, but they also reduce the workload of the programmer by allowing him to think about one case at a time, rather than all possible cases. In other words, examples allow a user to be concrete without being formal.

4.4.3 Interactive Programming is the Answer

A traditional programmer must write code that satisfies all possible inputs his program will encounter, while a Macho Programmer can consider each input individually. Macho therefore not only saves the programmer the work of writing code but also frees the programmer from difficult formal reasoning.

Ideally, however, the programmer would only be required to verify, not generate, concrete values. In this rosy scenario the programmer would input natural language and the system would offer a set of alternatives. The programmer could then reject incorrect cases, or suggest modifications, until eventually a correct program is negotiated. This is important because programming is not simply the act of transferring a mental vision into machine code. In reality, the requirements are fuzzy. Some things are more important than others, and still others can be waived or changed if they are difficult to implement. Interactive programming allows the programmer to take the path of least resistance to a satisfactory program.

Of course, this also requires considerably more accurate program synthesis from pure natural language, as well as much better understanding of general concepts, which no one really knows how to do at the moment.

4.4.4 Abstractions are Key

We believe that extending Macho to produce larger programs, while far from trivial, is not nearly as difficult or important as extending Macho to support more abstractions. Especially difficult are those that do not map precisely to calls or functions, like *first*, *last*, or *ignore*. These are especially important for interactive programming because they are often used to describe differences.

4.5 Related work

4.5.1 Natural-Language Processing

Programming by natural-language specification is an ambitious goal. Largely, Natural-Language Processing (NLP) does not aim at this goal presently because it depends on open questions in artificial intelligence, such as commonsense

reasoning [30, 47, 48], and knowledge-based natural-language understanding [15, 29, 63]. Instead, NLP works focus on fundamental semantics of natural language and computational applications.

The closest NLP works to ours, like *translation* and *question answering* [19], focus on using large corporas to yield reasonable results that are statistically reliable. Those applications rely on availability of simple solutions that do not require combination or deeper reasoning. For example, question answering looks for a sentence that includes the words that appear in the question.

Compared to NLP-based works, our work uses large corpora of possible interpretations of natural-language texts, applying combinatorial optimization to reach reasonable conclusions. In doing so, we use the type constraints available from our large repository of code segments together with available records of acceptable outputs (our unit tests). These enable combining results of natural-langauge-based segments in ways that are not possible in many other NLP applications.

4.5.2 Natural Language Programming

Compared to the thorny drudgery of formal languages, natural language seems easy and fun. Early researchers in computer science devoted extensive attention to talking to computers [7, 51, 65], but they soon realized that there were massive challenges. Natural language understanding is one of the hardest problems in AI, and one for which we still have no good solution. But even more fundamentally, those easy and fun descriptions are inherently abstract and ambiguous, leaving out key details which the system must infer. This creates difficulties not only for the system but for the user, who would not know which decisions the system made or even if the requested program made sense at all [21].

Despite these problems, by 1980 Ballard and Biermann had constructed a system capable of generating programs from basic natural language [6]. It worked by breaking down the input sentences grammatically and assuming that subjects and direct objects would be represented by objects while verbs would become functions: "Add row 1 to row 2" becomes $add(R1, R2)$. Their system had several hundred patterns and could handle relatively complicated sentences like:

```
Add the first and last positive entries in row 1 and the second to smallest entry
in the matrix to each entry in the last row.
```

Their key result was that English is flexible: the difficulties of natural language programming could be solved by restricting the input to precise, low-level language. But the straight-jacket they and their successors [12, 37, 54] placed on their users meant that their input language was essentially a verbose version of a simple programming language like Visual Basic.

4.5.3 Programming by example

Other methods of programming without a formal language apply learning from examples. Most of these [18, 28, 44] keep a database of sequences of user interface operations in a sort of GUI and try to match the current operations against it.

For many problems outside of user interface design these approaches are not practical. Operations such as listing a directory or opening an HTTP connection are not easy to describe by GUI, or require elaborate ad-hoc graphical design. Macho's unit tests require less information than examples of operations because they show only the initial and final state, and not the connecting path. This enables specification of procedures without giving intermediate details.

4.5.4 Code snippet search

Programming languages like Java or C# are more difficult to learn than their less bulky counterparts. This is partly because of the explosion of APIs in those languages. A true Java programmer must be familiar with thousands of classes, many of which change between versions.

Many researchers have proprosed tools to make it easier to find chunks of code. Prospector [45] leverages the type system to answer questions like "How do I go from an ICompilationUnit to an IDebugWindow object". SNIFF [9] works more on the natural language using the documentation and intersecting examples to determine the really critical function calls. Other attempts [27, 41, 56] use both, as well as the context of the function around the requested snippet. While snippet search is a useful tool, especially for novice programmers, there will often not be a library call that perfectly matches the users requirements.

4.5.5 Programming by sketching

Natural language is not always the most suitable way to express a computation. Programming by sketching [59, 60] works by taking a simple, easy to understand version of a very complex computation (often the inner loop from an encryption or signal processing algorithm) and generating a faster one from a sketch, a partially specified program. Sketching tries to make hard problems simple, while Macho tries to make simple problems trivial.

4.5.6 Deductive program synthesis

An alternative to programming that is still under development is to tell the computer what you want in a formal language. For simple mathematical properties, there are mechanical rules that can transform them into programs [46]. The more complicated the system, the trickier the synthesis and the larger the specification. Amphion [42] can synthesize programs that calculate properties of solar system objects using a Fortran library and graphical input.

Termite [55] generates device drivers automatically from a list of device and operating system state transitions. Like Macho, Bhansali *et. al* [4] tackle core utilities.

The problem with most of these is that because the user must still ultimately specify every detail, the formal description is not massively smaller than the code. For example, Termite's device-specific specification is approximately half the length of the corresponding Linux device driver C code, although probably somewhat less error-prone to describe. Macho allows the user to specify important details through examples while filling in the less important ones itself, thus achieving 90% of the results with 10% of the work.

4.6 Conclusions

In this paper we have discussed Macho, a system that synthesizes programs from a combination of natural language, unit tests, and a large database of source code samples. Macho is a simple proof of concept system, not yet directly useful for most programmers, but it can still synthesize basic versions of six small coreutils.

Chapter 5

Conclusions

Macho and Laika are examples of ways that Operating Systems can use machine learning to provide new and useful services which would either be too difficult or too time consuming to design or configure manually.

Laika shows how Operating Systems can provide new, higher-level abstractions to users. Using Artificial Intelligence techniques designed for face or speech recognition in the vastly simpler arena of computer systems is rather like using nuclear weapons to destroy an anthill.

Beyond pure natural language programming, Macho provides a blueprint for building flexible, informal computer interfaces. While touch screens or interruption management are useful techniques, the real problem with human computer interaction is the infuriating silicon combination of inflexible obstinacy and constant requests for trivial details.

I believe that machine learning in Operating Systems will prove to be a hot topic over the next decade, as computer systems finally advance beyond the stubborn formal obstinacy that has characterized them since their creation.

Appendix

Raw Code for Macho Programs

```
//"Print the first ten lines of a file.
//Lines Correct: 2/3
public static void Head1(java.lang.String p_file)
{
  //matches tail -n 1
  java.io.BufferedReader tmp = new BufferedReader(new FileReader(p_lines));
  tmp.readLine();
}

public static void FixedHead1(java.lang.String p_file)
{
  //matches tail -n 1
  java.io.BufferedReader tmp = new BufferedReader(new FileReader(p_lines));
  for(int i = 0; i < 10; i++)
    tmp.readLine();
}
```

Figure A.1: Code for Head

```
//"Copy src file to dest file."
//Lines Correct: 3/3
//Stitching Solution: 4
public static void Cp1(java.lang.String p_file1, java.lang.String p_file2)
{
  java.io.File tmp = new File(p_file1);
  java.io.File tmp_0 = new File(p_file2);
  copyFileRaw(tmp_0, tmp);
}

//"Copy file to file."
//Lines Correct: 3/3
//Stitching Solution: 4
public static void Cp2(java.lang.String p_file1, java.lang.String p_file2)
{
  java.io.File tmp = new File(p_file1);
  java.io.File tmp_0 = new File(p_file2);
  copyFileRaw(tmp_0, tmp);
}

//"Duplicate file to file."
//Lines Correct: 0/3
public static void Cp3(java.lang.String p_file1, java.lang.String p_file2)
{
  char tmp = '.'; //2
  char tmp_0 = '/'; //3
  p_file1.replace(tmp, tmp_0); //4
}
```

Figure A.2: Code for Cp

```
//"Sort the lines of a file."
//Lines Correct: 3/3
//Stitching Solution: 1
public static void Sort1(java.lang.String p_file)
{
  java.io.File tmp = new File(p_file);
  java.lang.String[] lines = ReadAllLines(tmp);
  Arrays.sort(lines);
}

//"Sort a file by line."
//Lines Correct: 3/3
//Stitching Solution: 2
public static void Sort2(java.lang.String p_file)
{
  java.io.File tmp = new File(p_file);
  java.lang.String[] lines = ReadAllLines(tmp);
  Arrays.sort(lines);
}

//"Sort a file."
//Lines Correct: 0/3
public static void Sort3(java.lang.String p_file)
{
  Arrays.sort(<UNSET java.io.File[]>)
}

//"Sort the contents of a file."
//Lines Correct: 0/3
public static void Sort4(java.lang.String p_file)
{
  //Hey, ls . . . .
  java.io.File tmp = new File(p_file);
  java.io.File[] contents = tmp.listFiles();
  java.lang.String[] tmp_0 = new String[contents.length];
  Arrays.sort(tmp_0);
}
```

Figure A.3: Code for Sort

```
//"Print the current working directory."
//Lines Correct: 3/3
//Stitching Solution: 29
public static void Pwd1()
{
  java.lang.String tmp = \"user.dir\";
  java.lang.String workingDirectoryCurrent = System.getProperty(tmp);
  System.out.println(workingDirectoryCurrent);
}

//"Print the current directory."
//Lines Correct: 3/3
//Stitching Solution: 11
public static void Pwd2()
{
  java.lang.String tmp = \"user.dir\";
  java.lang.String workingDirectoryCurrent = System.getProperty(tmp);
  System.out.println(workingDirectoryCurrent);
}

//"Print the user directory."
//Lines Correct: 3/3
//Stitching Solution: 24
public static void Pwd3()
{
  java.lang.String tmp = \"user.dir\";
  java.lang.String workingDirectoryCurrent = System.getProperty(tmp);
  System.out.println(workingDirectoryCurrent);
}

//"Print the working directory."
//Lines Correct: 0/3
public static void Pwd4()
{
  //breaks NLP; utter failure
}

//"Show the working directory."
//Lines correct: 0/3
public static void Pwd5()
{
  //no database results for show (working directory)
}
```

Figure A.4: Code for Pwd

```
//"Print the lines of a file."
//Lines correct: 5/5
//Stitching Solution: 1
public static void Cat1(java.lang.String p_file)
{
  java.io.File tmp = new File(p_file);
  java.lang.String[] lines = ReadAllLines(tmp);
  for(int tmp_1 = 0; tmp_1 < lines.length; ++tmp_1)  {
    java.lang.String tmp_0 = lines[tmp_1];
    System.out.println(tmp_0);
}

//"Read a file."
//Lines correct: 5/5
//Stitching Solution: 1
public static void Cat2(java.lang.String p_file)
{
  java.io.File tmp = new File(p_file);
  java.lang.String[] lines = ReadAllLines(tmp);
  for(int tmp_1 = 0; tmp_1 < lines.length; ++tmp_1)  {
    java.lang.String tmp_0 = lines[tmp_1];
    System.out.println(tmp_0);
}

//"Display the contents of a file."
//Lines correct: 0/5
public static void Cat3(java.lang.String p_file)
{
  //need input stream not output stream!
  new FileOutputStream(p_file);
}

//"Print a file."
//Lines correct: 1/5
public static void Cat4(java.lang.String p_file)
{
  //Well, at least it knows there is some printing going on.
  java.lang.String tmp = \".\";
  System.out.print(tmp);
}
```

Figure A.5: Code for Cat

```
//"Print the lines in a file matching a pattern."
//Lines Correct: 11/11
//Stitching Solution: 9
public static void Grep1(java.lang.String p_file1, java.lang.String p_pattern)
{
  java.io.File tmp = new File(p_file);
  java.lang.String[] lines = ReadAllLines(tmp);
  for(int tmp_1 = 0; tmp_1 < lines.length; ++tmp_1)  {
    java.util.regex.Pattern tmp_2 = Pattern.compile(p_matching);
    java.lang.String tmp_0 = lines[tmp_1];
    boolean matching = tmp_2.matcher(tmp_0).find();
    if(matching) {
      System.out.println(tmp_0);
    }  else  {
    }
  }
}

//"Find a pattern in the lines of a file."
//Lines Correct: 7/11
public static void Grep2(java.lang.String p_file1, java.lang.String p_pattern)
{
  java.io.File tmp = new File(p_file);
  java.lang.String[] lines = ReadAllLines(tmp);
  int tmp_0 = lines.length;
  boolean[] tmp_1 = new boolean[tmp_0];
  for(int tmp_5 = 0; tmp_5 < lines.length; ++tmp_5)  {
    java.util.regex.Pattern tmp_3 = Pattern.compile(p_pattern);
    java.lang.String tmp_4 = lines[tmp_5];
    boolean tmp_2 = tmp_3.matcher(tmp_4).find();
    tmp_1[tmp_5] = tmp_2;
  }
}

//"Search file for a pattern."
//Lines Correct: 1/11
public static void Grep3(java.lang.String p_file1, java.lang.String p_pattern)
{
  Pattern.compile(p_file); //4531
}
```

Figure A.6: Code for Grep

```
//"Print the names of the files in a directory.  Sort the names."
//Lines Correct: 17/17
//Stitching Solution: 5
public static void Ls1(java.lang.String p_directory) {
  java.io.File tmp = new File(p_dir);
  java.io.File[] files = tmp.listFiles();
  boolean tmp_3 = tmp.isDirectory();
  if(tmp_3) {
    Arrays.sort(files);
    for(int tmp_1 = 0; tmp_1 < files.length; ++tmp_1)  {
      java.io.File tmp_0 = files[tmp_1];
      java.lang.String names = tmp_0.getName();
      boolean tmp_2 = tmp_0.isHidden();
      if(tmp_2) {
      }  else  {
        System.out.println(names);
      }
    }
  }  else  {
    System.out.println(tmp + "");
  }
}

//"Print the contents of a folder. Sort."
//Lines Correct: 17/17
//Stitching Solution: 43
public static void Ls2(java.lang.String p_folder) {
  java.io.File tmp = new File(p_folder); //15097
  java.io.File[] contents = tmp.listFiles(); //15098
  boolean tmp_3 = tmp.isDirectory();
  if(tmp_3) {
    Arrays.sort(contents);
    for(int tmp_1 = 0; tmp_1 < contents.length; ++tmp_1)  {
      java.io.File tmp_0 = contents[tmp_1];
      java.lang.String tmp_2 = tmp_0.getName();
      boolean tmp_4 = tmp_0.isHidden();
      if(tmp_4) {
      } else {
       System.out.println(tmp_2);
      }
  }
  else {
    System.out.println(tmp + "");
  }
}
```

Figure A.7: Code for ls-1

```
//"Print the entries of a directory."
//Lines Correct: 0/17
public static void Ls3(java.lang.String p_folder) {
  //Thinks entries means directory is zipped
  java.lang.String tmp = \"META-INF/MANIFEST.MF\"; //5507
  java.io.File files = new File(p_directory); //5508
  java.util.zip.ZipFile tmp_0 = new ZipFile(files); //5509
  java.util.zip.ZipEntry entries = tmp_0.getEntry(tmp); //5510
  java.lang.String tmp_1 = entries.getName(); //5511
  System.out.println(tmp_1); //5512
}

//"Print the files in a directory
//Lines Correct: 5/17
public static void Ls4(java.lang.String p_folder)
{
  //Synthesis can't fix it because without the sorting the output
  //is considered too far off.
  java.io.File tmp = new File(p_directory);
  java.lang.String[] files = tmp.list();
  for(int tmp_1 = 0; tmp_1 < files.length; ++tmp_1)  {
    java.lang.String tmp_0 = files[tmp_1];
    System.out.println(tmp_0);
  }
}
```

Figure A.8: Code for ls-2

```
//"Download File."
//Lines Correct: 0
public static void Wget1(java.lang.String p_file1)
{
  //database failure!  Candidates aren't sufficiently frequent.
}


//"Open Url. Read lines."
//Lines Correct: 0
public static void Wget2(java.lang.String p_url)
{
  java.net.HttpURLConnection tmp = (HttpURLConnection)new
                      URL(p_url).openConnection();
  java.io.BufferedReader tmp_0 = new BufferedReader(new
               InputStreamReader(tmp.getInputStream()));
  tmp_0.readLine();
}
```

Figure A.9: Code for Wget

```
//"Print the lines of a file.  Ignore adjacent lines"
//Lines correct: 5/8
public static void Uniq1(java.lang.String p_file)
{
  //same as cat
  java.io.File tmp = new File(p_file);
  java.lang.String[] lines = ReadAllLines(tmp);
  for(int tmp_1 = 0; tmp_1 < lines.length; ++tmp_1)  {
    java.lang.String tmp_0 = lines[tmp_1];
    System.out.println(tmp_0);
}

public static void FixedUniq1(java.lang.String p_file)
{
  //same as cat
  java.io.File tmp = new File(p_file);
  java.lang.String last = "234u8932892";
  java.lang.String[] lines = ReadAllLines(tmp);
  for(int tmp_1 = 0; tmp_1 < lines.length; ++tmp_1)  {
    java.lang.String tmp_0 = lines[tmp_1];
    if(tmp_0 != last)
      System.out.println(tmp_0);
    last = tmp_0;
}
```

Figure A.10: Code for Uniq

References

[1] 2007 malware report: The economic impact of viruses, spyware, adware, botnets, and other malicious code. http://www.computereconomics.com/page.cfm?name=Malware%20Report.

[2] Clamav website. http://www.clamav.org.

[3] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *In Comp. Construct*, pages 5–23. Springer-Verlag, 2004.

[4] S. Bhansali and M. T. Harandi. Synthesis of unix programs using derivational analogy. *Mach. Learn.*, 10(1):7–55, 1993.

[5] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*. USENIX, Aug. 2005.

[6] A. W. Biermann and B. W. Ballard. Toward natural language computation. *Comput. Linguist.*, 6(2):71–86, 1980.

[7] D. Bitzer and D. Skaperdas. Plato iv - an economically viable large scale computer-based education system. In *National Electronics Conference*, 1968.

[8] D. Bruschi, L. Marignoni, and M. Monga. Detecting self-mutating malware using control flow graph matching. Technical Report, Universitaa degli Studi di Milano, http://idea.sec.dico.unimi.it/ lorenzo/rt0906.pdf.

[9] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for Java using free-form queries. In *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 385–400, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] P. M. Chen and B. D. Noble. When virtual is better than real. In *HotOS*, pages 133–138. IEEE Computer Society, 2001.

[11] D. M. Chess and S. R. White. An undetectable computer virus. In *Proceedings of the 2000 Virus Bulletin Conference*, 2000.

[12] R. Chitchyan, A. Rashid, P. Rayson, and R. Waters. Semantics-based composition for aspect-oriented requirements engineering. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 36–48, New York, NY, USA, 2007. ACM.

[13] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)*, pages 32–46, Oakland, CA, USA, may 2005.

[14] F. Cohen. Computer viruses: Theory and experiments. In *Computers and Security*, pages 22–35, 1987.

[15] M. Connor, Y. Gerner, C. Fisher, and D. Roth. Minimally supervised model of early language acquisition. In *Annual conference on computational natural language learning (CoNLL)*, 2009.

[16] J. C. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. *ACM Trans. Softw. Eng. Methodol.*, 9(1):51–93, 2000.

[17] A. Cozzie. An introduction to libertarianism, or my government is run by cretins. http://www.acoz.net/economics/.

[18] A. Cypher. Eager: programming repetitive tasks by example. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 33–39, New York, NY, USA, 1991. ACM.

[19] R. de Salvo Braz, R. Girju, V. Punyakanok, D. Roth, and M. Sammons. An inference model for semantic entailment in natural language. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2005)*, 2005.

[20] D. E. Denning. An intrusion-detection model. *IEEE Trans. Softw. Eng.*, 13(2):222–232, February 1987.

[21] E. W. Dijkstra. On the foolishness of 'natural language programming'. http://userweb.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD667.html.

[22] J. Franklin, V. Paxson, A. Perrig, and S. Savage. An inquiry into the nature and causes of the wealth of internet miscreants. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 375–388, New York, NY, USA, 2007. ACM.

[23] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*. The Internet Society, 2003.

[24] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, New York, NY, USA, 1996. ACM.

[25] D. Goodin. Move over storm - there's a bigger, stealthier botnet in town. http://www.theregister.co.uk/2008/04/07/kraken_botnet_menace/.

[26] J. Gordon. Lessons from virus developers: The beagle worm history through april 24, 2005. In *SecurityFocus Guest Feature Forum*, 2004.

[27] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A search engine for finding highly relevant applications. In *ICSE '10: Proceedings of the International Conference on Software Engineering 2010*, 2010.

[28] D. C. Halbert. *Programming by example*. PhD thesis, University of California, Berkeley, 1984.

[29] J. R. Hobbs. Deep lexical semantics. In *9th international conference on intelligence text processing and computational linguistics (CICLing-2008)*, 2009.

[30] J. R. Hobbs and R. C. Moore, editors. *Formal Theories of the Commonsense World*. Greenwood Publishing Group Inc., Westport, CT, USA, 1985.

[31] I. Hsi, C. Potts, and M. Moore. Ontological excavation: Unearthing the core concepts of applications. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, 2003.

[32] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the 2006 USENIX Annual Technical Conference: May 30–June 3, 2006, Boston, MA, USA*, 2006.

[33] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 14–24, New York, NY, USA, 2006. ACM Press.

[34] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 91–100, New York, NY, USA, 2008. ACM.

[35] M. Jordan. Dealing with metamorphism, October 2002.

[36] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 2005 Symposium on Operating Systems Principles (SOSP)*, pages 91–104, October 2005.

[37] R. Knöll and M. Mezini. Pegasus: first steps toward a naturalistic programming language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 542–559, New York, NY, USA, 2006. ACM.

[38] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables, 2005.

[39] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 91–100, Tucson, AZ, December 2004.

[40] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. 2003.

[41] G. Little and R. C. Miller. Keyword programming in java. In *ASE '07: Proceedings of the 22nd International Conference on Automated Software Engineering*, pages 84–93, New York, NY, USA, 2007. ACM.

[42] M. R. Lowry and J. V. Baalen. Meta-amphion: Synthesis of efficient domain-specific program synthesis systems. *Autom. Softw. Eng.*, 4(2):199–241, 1997.

[43] J. Ma, J. Dunagan, H. J. Wang, S. Savage, and G. M. Voelker. Finding diversity in remote code injection exploits. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 53–64, New York, NY, USA, 2006. ACM.

[44] P. Maes and R. Kozierok. Learning interface agents. In *AAAI*, pages 459–465, 1993.

[45] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI '05: Proceedings of the 2005 Conference on Programming Language Design and Implementation*, pages 48–61, New York, NY, USA, 2005. ACM.

[46] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Trans. Softw. Eng.*, 18(8):674–704, 1992.

[47] C. Matuszek, M. Witbrock, R. C. Kahlert, J. Cabral, D. Schneider, P. Shah, and D. Lenat. Searching for common sense: Populating cyc from the web. In *In Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 1430–1435, 2005.

[48] J. McCarthy. Notes on formalizing context. In *IJCAI*, pages 555–562, 1993.

[49] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[50] J. Nugent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Controlling your place in the file system with gray-box techniques. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003.

[51] S. R. Petrick. On natural language based computer systems. *IBM J. Res. Dev.*, 20(4):314–325, 1976.

[52] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot–a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 2004 USENIX Security Symposium*, August 2004.

[53] M. Polishchuk, B. Liblit, and C. W. Schulze. Dynamic heap type inference for program understanding and debugging. *SIGPLAN Not.*, 42(1):39–46, 2007.

[54] D. Price, E. Riloff, J. Zachary, and B. Harvey. Naturaljava: a natural language interface for programming in java. In *IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces*, pages 207–211, New York, NY, USA, 2000. ACM.

[55] L. Ryzhyk, P. Chubb, I. Kuz, E. L. Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, USA, Oct 2009.

[56] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. *SIGPLAN Not.*, 41(10):413–430, 2006.

[57] P. K. Singh and A. Lakhotia. Analysis and detection of computer viruses and worms: An annotated bibliography. In *ACM SIGPLAN Notices*, pages 29–35, 2002.

[58] M. Sivathanu, V. Prabhakaran, F. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, CA, March 2003.

[59] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *PLDI '05: Proceedings of the 2005 Conference on Programming Language Design and Implementation*, pages 281–294, New York, NY, USA, 2005. ACM.

[60] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415, New York, NY, USA, 2006. ACM.

[61] B. Steensgaard. Points-to analysis by type inference of programs with structures and union. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 136–150, 1996.

[62] A. Sutton and J. Maletic. Mappings for accurately reverse engineering uml class models from c++. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE)*, 2005.

[63] K. Toutanova and C. D. Manning. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Joint SIGDAT conference on empirical methods in natural language processing and very large corpora (EMNLP/VLC-2000)*, pages 63–70, 2000.

[64] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[65] J. Weizenbaum. Eliza–a computer program for the study of natural language communication between man and machine. In *Communications of the ACM*, 1966.

[66] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, 1995.

[67] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. *Proceedings of the 2002 IEEE International Conference on Data Mining*, 2002.

[68] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.

[69] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 272–281, New York, NY, USA, 2006. ACM.

Author’s Biography

Anthony Cozzie is a graduate student at the University of Illinois at Urbana-Champaign. While he usually lives a gnomish, underground existence lit only by the pallid glow of LCD monitors, he does manage to sneak away from time to time and play basketball, read Shakespeare, and watch anime.