STATIC DETECTION OF DISASSEMBLY ERRORS

by

Nithya Krishnamoorthy

A Thesis Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

In the Graduate College

THE UNIVERSITY OF ARIZONA

2 0 1 0

FINAL EXAMINING COMMITTEE APPROVAL FORM

Replace this page with the correct approval form.

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: ______________________________

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

Saumya Debray Professor, Department of Computer Science	Date

DEDICATION

To my brother, Prasanna, who makes me reach for my best self and the stars. And to my parents. For everything.

ACKNOWLEDGEMENTS

I thank Dr. Saumya Debray for his encouragement and guidance throughout my two years here. I walked away from each of our meetings energized and enlightened. I have immensely benefited from and will forever cherish my interactions with him.

I thank Dr. Ian Fasel and Dr. Christian Collberg for being on my committee and providing valuable comments on my thesis drafts.

I am grateful to my mentors, Dr. Ramanujam and Balasubramanian for showing the way. I am where I am because of them.

I am indebted to all my friends, old and new, for keeping me going these two years and for the wonderful times that we have shared. Special thanks to Anju, Archa and Aravind, for doubling as the best on-call psychiatrists when I needed it most; Somu, for all the help throughout my two years in Tucson, starting from the very first day at the airport to all the linux tips in the lab to reading my thesis drafts; Raquel, Kate, Sahana, Nassim, Farnaz and Jinyan for being the best bunch of grad girls; Ashwini, Kartik, Pandian and Arvinth for making Tucson feel like home.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The first step in understanding the semantics of a binary executable is to extract the assembly instructions that could get executed if it is allowed to run. This sequence of assembly instructions, typically obtained by static disassembly, is assumed to be correct by many analysis techniques that build on it. However, static disassembly can be incorrect; there can be accidental errors during disassembly or a disassembler can be deliberately misled by binary obfuscation techniques, rendering this assumption invalid.

This thesis proposes a machine learning approach to statically identify disassembly errors in a static disassembly, so that such potential errors can be examined more closely, using, for example, dynamic analysis. We show that a decision tree classifier that is built using this approach identifies most known disassembly errors in the malware that we used for evaluation.

CHAPTER 1

Introduction

Static Disassembly is the process of retrieving the assembly level instruction sequence from a given binary executable. It is the first step in reverse engineering an executable, with the goal of understanding the program's high level semantics when the source code is unavailable (Christodorescu and Jha, 2003; Kruegel et al., 2004; Singh et al., 2003; Balakrishnan and Reps, 2004; Balakrishnan et al., 2005; Christodorescu et al., 2005; Prasad and Chiueh, 2003; Xu et al., 2000). A security researcher, when confronted with a suspicious binary, will first abstract away from the binary to the assembly level and then work on this representation for gaining an understanding of its functionality. An extensive body of security research thus depends on the disassembly to be correct. But this is not always true.

Accidental disassembly errors are commonly caused by the presence of legitimate data such as jump tables within the code stream - a disassembler will misidentify this data as instruction bytes and disassemble them. Since these are data bytes, the assembly code produced by disassembling them will never get executed. Disassemblers can also be deliberately misled by binary obfuscation (Linn and Debray, 2003; Popov et al., 2007). These obfuscators, unlike many others that target the decompilation phase of reverse engineering, where assembly code is transformed to a higher level representation, target disassembly which is the very first phase of reverse engineering. They quite easily trick the disassembler and effectively thwart all the analysis that take the output of the disassembler as their input.

Identifying erroneous disassembly is also not easy. Many times, disassembly errors go unnoticed, with the disassembler exiting without a warning. This is especially true in the widely used Intel IA-32 architecture because of two main reasons.

One, the encoding density of the instructions is very high. This means that most byte sequences will disassemble into valid assembly instructions. An example that illustrates this phenomenon is a malware (Hybris.c.exe) in which almost the entire binary is in encrypted state initially and an automatic unpacker of less than 10 instructions does the decryption before transferring control to the modified section of the code. When the binary is disassembled though, with the disassemble-all flag using objdump (Objdump, 2005), less than 4% of the bytes are actually identified as invalid, even though more than 99.6% of the disassembled instructions do not correspond to any instruction that will get executed, and some of them are, in fact, encrypted bytes of the actual instructions.

Two, typical of a CISC architecture, the instructions are of variable length. The IA-32 instructions can be anywhere between 1 byte and 15 bytes long. This makes tricking the disassembler easy because any instruction can start at any byte address within the binary since there are no fixed instruction boundaries. It is this feature of CISC architecture, along with the presence of branch instructions that make the problem of obtaining a precise static disassembly from an arbitrary input binary an undecidable problem (Schwarz et al., 2002).

One way to get a correct disassembly is to execute the binary and observe the instructions that get executed, however this type of dynamic analysis will provide only a single execution path through the program which may not be the same instruction sequence if the same program is run under different environmental conditions. For instance, a particular malware executable may never execute its malicious payload on any day other than 13th, Friday. The execution trace of this piece of binary will never provide any evidence of its malicious intent if it is obtained on any other day of the year. Even though it is possible to force execution through many paths of the program (Moser et al., 2007) to gain more insight into the working of the binary, this may not be a practical solution. Finding the right conditions for forcing execution in a certain direction could be very hard and the number of possible paths can be very large.

This thesis proposes a machine learning approach to identify regions that may contain disassembly errors, given a static disassembly of any binary. This is based on the approach taken by malware analysts who painstakingly look at assembly instructions to try to identify which parts of the disassembly could be incorrect, which they seem to be able to tell by looking closely at instructions and spotting "weird" patterns. For example, some constant operand to an instruction might be too large, or some mnemonic might be uncommon, or a particular combination of operations may just not result in predictable behavior. These incorrect disassemblies are statistically quite different from correct ones and this is the basis for using the machine learning approach. We carefully pick indicators of incorrect disassembly as representations of the disassembled instruction. These are then extracted from many correct and incorrect disassemblies and a decision tree classifier is trained. We see that the classifier obtained works quite accurately on test files that are similar to our training files. With arbitrary malware, the classifier does well in identifying the erroneous areas of the disassembly, but does not do so well in identifying the correct disassembly. Many correctly disassembled areas are falsely classified as incorrect.

CHAPTER 2

Disassembly Errors and Machine Learning

2.1 Disassembly Errors

Reverse Engineering is the process of discovering a high level abstraction of the binary executable whose source is unavailable. This is generally the case for proprietary software as well as malware. It is therefore necessary from a security perspective to reverse engineer a suspect binary to find out what it actually does. Since reading and understanding machine code is not an option for most humans, security researchers try to take the binary to higher levels of abstractions that can be more easily understood: they extract equivalent assembly code, create control flow graphs, discover the data flow in the program, or seek to create a version of the source code. However this process of abstraction is not easy. Malware writers and owners of proprietary code deliberately obfuscate code in various ways to thwart reverse engineering to conceal the logic of their binaries. Anti-disassembly is one such technique by means of which the disassembler is tricked into producing an incorrect assembly instruction sequence that does not represent the actual functioning of the binary, if it is allowed to run. An example of this form of obfuscation is described below and its implications explained.

Figure 2.1 is the static disassembly of Win32.Microjoin.R whose executable was packed with the software protection tool Aspack(asp, 2009). The disassembly algorithm used here, recursive disassembly, starts at the entry point (available in

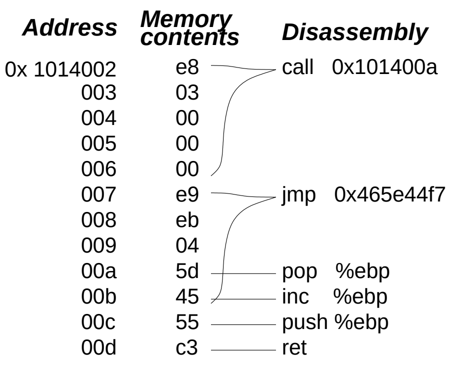


Figure 2.1: Static disassembly

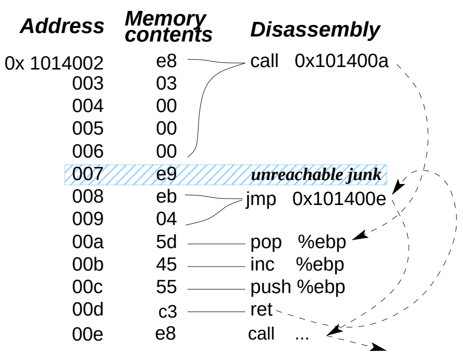


Figure 2.2: Actual instruction sequence

the header of the executable) and recursively disassembles at all target addresses of every branch and jump instruction. In this example the disassembly consists of just six instructions because one of the target addresses is outside the address range of the binary obtained from the section header of the executable.

An executable is divided into different memory regions, called sections, each of which typically requires different protection levels. For example, the code section is usually read/execute, the data section is read/write. Each of these regions must be aligned to a page boundary prior to execution, so that the correct protection levels can be granted to the appropriate pages. However, in order to save space, the sections are not page aligned on disk. It is therefore the task of the linker to read from the header of the executable and map the different sections to different memory regions individually. The section header of the executable contains information about the different sections and the addresses to which they should map. This section header information is therefore considered to be the extent of the knowledge that the compiler can assume when writing the binary and usually no more assumptions are made about any other memory regions. This particular binary had 5 sections with addresses ranging from 0x1001000 to 0x101c000 according to the section header. Anything beyond these addresses cannot be assumed to be available during execution and thus cannot be used as an absolute address when the instruction sequence is generated by the compiler. During disassembly, on the other hand, target addresses, like 0x465e44f7, that fall outside this range cannot be recursively disassembled at. Hence disassembly stops after six instructions are disassembled. These six instructions do close to nothing and the binary appears benign.

However this static disassembly is far removed from reality. The actual instruction sequence that gets executed is shown in Figure 2.2. It can be seen that the packer uses the "branch function" technique and junk byte insertion (Linn and Debray, 2003) to confuse the disassembler into producing an incorrect representation of the binary. The pop - increment - push sequence changes the return address of the function that has been called to be one byte beyond what it was originally. This technique, to jump to a predetermined target address using a function call/ return construct is called the "branch function". This simple manipulation is enough for making the disassembler miss the jump instruction at 0x1014008 that jumps to address 0x101400e which eventually leads to the unpacking and execution of the malicious payload.

The anti-disassembly obfuscation shown in this example effectively leads to much of the code being missed by the disassembler and also any high level analysis that could have been done above it. Another point to note is that the disassembly that was obtained was itself valid - there was no obvious indication that there was an error during the disassembly. These are some of the reasons because of which identification of disassembly errors in a large file is a hard problem.

It is in fact sufficient to conceal very few key instructions to hide malicious intent. A simple decryptor loop can be written with as few as 10 instructions and each of these instructions and the control transfer between them can be carefully hidden in the code section by obfuscation as shown. The decrytor can then decrypt data available in a data section that is not marked as containing code and then transfer control to the newly decrypted instructions. This way, the entire code section could be disassembled, but the malicious code, encrypted and hidden within a data section would be missed completely by the disassembler. And since even one missed instruction can cause an invalid disassembly with serious consequences, as shown in this example, it is important to identify these disassembly errors with high precision.

A tedious way to identify disassembly errors in obfuscated binaries is to manually examine a disassembly and watch for suspicious characteristics. Security researchers who do this will then go on to trace into the suspicious portions of the disassembly to obtain the actual instruction sequence.

However, this example shows some signs that the disassembly may be incorrect/ incomplete. The most striking is a jump to an address that is not present in the code, a sure sign of something being amiss. Another suspicious instruction is a call to a middle of another instruction, but this could be the working of a compiler that does clever optimization for code size. Also, most of the 52K binary was not disassembled, this is a cause for suspicion but this is not always the case in incorrect disassemblies.

The basis of this thesis is this observation that it is common for incorrect disassemblies to exhibit some properties that make them appear suspect and that correct disassemblies usually do not have these characteristic traits. A machine learning approach is then deployed to identify the errors.

2.2 Supervised Machine Learning

Supervised machine learning is a learning technique where the learner approximates a function mapping an input onto a set of classes by looking at input/ output examples of the function. For the approximated function to be able to correctly predict the class containment of future inputs, the features and statistical distribution of the features in the examples should reflect, as accurately as possible, the properties on which containment in the distinct classes is dependent upon. However, since the training inputs are finite and the future is indefinite, these machine learning algorithms do not guarantee correct classification in all cases, although they may provide probability bounds for the performance of the algorithm.

Our training set consists of a set of pairs ($<x_1$, x_2, x_3, ... $x_n>$, f($<x_1$, x_2, x_3, ... $x_n>$)) where x, the input vector, is the description of the disassembly and it falls into either the correct or the incorrect disassembly class, the value of function f which needs to be learned. For our purposes, the training input consists of positive examples, i.e., pairs of the form ($<x_1$, x_2, x_3, ... $x_n>$, correct) where vector x is a representation of a correctly-disassembled code sequence, and negative examples, i.e., pairs of the form ($<x_1$, x_2, x_3, ... $x_n>$, incorrect) where vector x is a representation of an incorrectly-disassembled code sequence. The desired output is a classifier that is able to distinguish between correct and incorrect disassemblies. As long as we are able to find a good representation for the disassembly, a large number of training examples, and our unknown future inputs behave in a manner similar to the training set, we get pretty accurate results on them.

2.3 Decision Tree Classifiers

Decision tree classifiers are a type of machine learning classifiers that are simple and easy to understand. Given a training set of the form (<x_1, x_2, x_3, ... x_n>, y) a decision tree is constructed by a learning algorithm that splits the training examples into subsets based on values of a specific feature xi. This process is recursively repeated on each of the subsets until all the elements of the subset belong to the same class y. The result is a tree that has features as its interior nodes and possible values of those feature as edges to their children. The leaves of the tree are the values of the class y. This decision tree can then be used to find the class for any feature vector by traversing the path from the root to leaf, taking the edges that correspond to the input variables.

Decision trees are usually easy to understand and the decisions made can be validated and analysed. This ease of use was one of the main reasons we chose to use decision tree classifiers as our learning technique. Many times during the course of this research, the reasoning (path from root to leaf) for the incorrect decisions that the early trees made, has given key information about what was going on and which feature we must look into more closely to get better results. For example, we came to know that for jump instructions the earliest trees did not require details about the operand to make a decision. Intuitively, this was unexpected. Also, the classification was incorrect in many of these cases. Adding a feature that represented the validity of the target address and training on this new set of features made the tree look much more reasonable. This tree was evaluating more features before a decision was made for jump instructions and also gave better results.

We use C4.5, a decision tree package that uses Quinlan's ID3 algorithm (Quinlan, 1986). Though constructing an optimal decision tree is NP-hard, this algorithm uses a greedy heuristic to minimize the size of the tree. The particular heuristic used is based on the information theory concept of entropy - the learning algorithm tries to greedily maximize information gain at each step by choosing as the interior node, the feature that yields maximal information gain at that stage of the algorithm.

CHAPTER 3

Decision Trees for Identifying Disassembly Errors

3.1 Data collection

For the construction of decision trees for our classification problem, we need correct and incorrect disassemblies in large numbers, so that we can extract the appropriate features from these for our training set. For this we need to identify a sufficiently large set of instructions that have been correctly disassembled. Given an arbitrary binary without any additional information, it has been established in the previous chapters that, it is impossible in general to statically obtain its "correct" disassembly. In fact, this is exactly the larger problem that we are trying to solve. Dynamic multipath tracing (Moser et al., 2007) is not a practical alternative because it is very time consuming and a single path would explore only a small fraction of the code and that would not provide us with enough data for training a classifier that will give good results.

A different approach to gathering training data is taken. We use gcc compiled binaries that have symbol table and relocation information, which are sufficient for accurate static disassembly of the binary. By default, executables do not contain this information, even if they do, they cannot be relied upon to be accurate, especially in suspicious executables which are our target set. The relocation tables that can be produced during compilation contains information about all the addresses that need to be changed if the loader relocates the program. This includes indirect jumps and calls which can therefore can be determined accurately from the relocation tables. Ofcourse, this level of detail is not expected to be available in the binaries whose disassemblies we wish to analyze, it is only for purposes of training that we use this data.

Since we know the correct disassembly, the list of addresses at which any correct disassembly starts is computed. We call this the InstAddr list. All addresses that belong to the code section of the binary but do not belong to this set are added to the OffsetAddr list. We then disassemble the binary starting at every address in its code section. The training sets are obtained as follows: instructions that are obtained by starting disassembly at any of the InstAddr list are added to the positive set and the those that start at the OffsetAddr list are added to the negative set.

Typically, binary obfuscation is done by adding junk bytes in such a way that the disassembly at these bytes overlap with the correct disassembly. This causes the disassembler to miss the address at which the correct instruction starts. An example was the five byte long, incorrect instruction disassembled at address 0x1014007, which overlapped with the actual instructions at addresses 0x1014008, 0x101400a and 0x101400b and therefore missed. Incorrect disassembly tends to overlap with the correct ones, justifying the way the incorrect training set is created.

The test data that we used were binaries that were deliberately obfuscated to cause disassembly errors. For evaluation of the performance of the decision tree classifier, we need to know the the correct disassembly, in other words the"ground truth". For the benchmark tests, this was obtained from the obfuscator which was made to output the list of actual addresses once it wrote out the obfuscated binary. These binaries were disassembled using the GNU objdump (Objdump, 2005) utility and were used for evaluation.

We also evaluated our decision trees on malware. The challenge here was to identify the ground truth and the time consuming approach we took was to execute the binary and collect the execution trace using Ollydbg v2.0-Beta 2 Final (Yuschuk, 2009). Even though we were able to get actual addresses in very small numbers, because the execution was along a single code path, the number of offset addresses, especially in self modifying malware, were considerable, thus helping us evaluate the classifier on the many incorrect disassemblies.

3.2 Training set properties

3.2.1 Importance of feature vector identification

The identification of a good vector representation for an instruction that has been disassembled forms the bulk of this work. Once we have a good representation of the disassembly that conveys all the significant properties of the instruction, which includes those properties that differentiate most correct and incorrect disassemblies, the learning algorithm will be able to produce a decision tree than can predict the correctness of disassemblies quite accurately.

One easy way to do this is to just feed the binary bits that make the instruction, to the decision tree. Each instruction can then be mapped to a natural number, a perfectly valid representation of the disassembly - in fact the same representation that the processor works with! But this representation does not capture the essence of the disassembly, it is just the syntax of the instruction in machine level representation. The disassembly, as we know it, is an instruction that imparts information about an operation to be performed by the processor, that works with its operand in certain ways and writes the output to a certain location. Any of these could have specific characteristics that could could be unusual. It is not just the bits that we know about in a disassembled instruction - we know much more, and this abstraction from bits is part of the knowledge that can aid the identification of the correctness of the instruction sequence.

One other way to form the input to the decision tree construction algorithm, is directly pulling the text of the disassembled instruction, but neither would that be useful for our purposes. This is the case because the actual string/ text of the disassembly does not really predict the validity of the instruction that has been decoded as much as the various parts of it do, for instance the mnemonic is a fine indicator - a standard compiler will typically not generate certain assembly instructions. The IA-32 manual (Intel Corp., 2008) reveals that certain combinations of mnemonic and prefixes to be invalid. An invalid address as a target of a jump is most certainly invalid disassembly or at least code that is illegal in some way.

It is the identification of those aspects of a disassembly that are significant to the classification of correct disassembly from the incorrect that is the interesting work involved in using a machine learning learning approach to solve our problem. This way, the decision tree construction algorithm can learn statistically significant patterns from samples and apply them on unseen inputs to infer which parts of a disassembly may be incorrect. In fact, we strongly feel that given the right indicators to work with, any dependable statistical classifier can do a reasonable job at determining the accuracy of the disasssembly.

3.2.2 Statistical distribution of the legal and illegal training sets

During our initial attempts at using decision trees to solve this problem, when we used a feature set that was directly from the structure of an IA-32 instruction, the decision tree constructed tended to classify most instructions as invalid. We inferred that this was because the number of unique legal and illegal disassemblies that we had in our training set was very disproportionate. The number of actual addresses were about 4 times smaller than the number of offset addresses in the training files. Therefore this training set suggested that it was highly likely that any given instruction was invalid and hence the decision tree that was learned from this classified all instructions as illegal.

To get around this, we experimented with limiting the number of valids and invalids to a manageable size of 5000 records each. This did not work for two main reasons. One, the number of valids and invalids that were being discarded had crucial information that we were ignoring. Two, hidden in this approach was the information that the probability of an instruction being valid or invalid was equally likely. We tried to solve the second issue by changing the decision tree code to mark an instruction as invalid if it had not seen it before. This aggravated the impact of the first issue. The many valid disassemblies that we were discarding had key information about valid disassemblies that we were now classifying as invalid just because they were unseen in the training set. To solve this, we took the approach of keeping all the valids and invalids in the training set and then adding in the valid set repeatedly into the pool until the total number of valids became greater than or equal to the number of invalids. This approach finally gave us the required behavior from the decision tree construction algorithm. Even though we were still suggesting that the valids and invalids were equally likely, the fact that we did not throw away any data we had collected for this helped and we started seeing reasonable results.

3.3 Limitations of approach

Supervised machine learning algorithms assume that the future is similar to the past, in that the unseen inputs have the same significant properties as those in the training set. But in the case of malware, code that looks weird may not be because of a disassembly error. It could just be because of a programmer who writes in assembly, using constructs that are not used by standard compilers, optimizing code in ways that compilers wouldn't, or wasting processor cycles just to send researchers on a red herring chase. This makes malware reverse engineering a moving target because malware writers can hand tune their code to work in unexpected ways and break assumptions that are made by the reverse engineering techniques. In our case, the assumption is that any disassembly that has properties similar to the properties of incorrect disassembly in the training set is also an incorrect disassembly. But this need not be the case for hand written or obfuscated binaries.

In several malware samples we come across actual instruction sequences that standard compilers would typically not generate. Typically this was either dead code being inserted just to increase the bulk of the disassembly, to cause distress to reverse engineers, or obfuscations that bloated a simple instruction into a big set of complex instructions that included many abnormal immediate values and computational steps.

```
sub eax 0x5590e07d // eax = eax - 0x5590e07d          push 0x7659fea9
push ecx
push 0x196f3deb
pop ecx
pop esi
pop edi
add eax, 0x70c3db72 // eax = eax + 0x70c3db72
mov cl, 0xa8
```

which is the same as the following sequence:

```
mov esi, ecx
mov ecx, 0x196f3da8
mov edi, 0x7659fea9
add eax, 0x1b32faf5
```

Indeed, almost any operation that can be accomplished with a single instruction can be also be written using an arbitrarily long instruction sequence that finally results in the same behaviour. And some of these instructions can manipulate values that will never be used. When the decision tree comes across such cases, many of the correctly disassembled but incorrect looking instructions classified as incorrect, causing the false illegal rate to go very high for some malware samples. This is a limitation that we have been unable to deal with.

CHAPTER 4

Feature Set Identification

4.1 Feature Set Identification

In order to construct a decision tree for static disassembly, we have to identify which features of the disassembled instructions are relevant to identifying potentially erroneous disassemblies, and what values each of these features can take. This section discusses the various features we considered and what worked, did not work and why.

We considered two metrics for evaluating the performance of the decision trees that were built during each stage of the evolution of the feature vector: false legals and false illegals. Intuitively, "false legal" refers to situations where the classifier erroneously classifies some part of a disassembly as being correctly disassembled where in reality there is a disassembly error; "false illegal" refers to the opposite situation, where the classifier erroneously indicates a disassembly error for a part of the program that has been correctly disassembled.

4.1.1 Instruction Feature Evolution

The selection of a set of features used to construct a decision tree classifier is a process that requires careful tuning. This section discusses the evolution of the feature set we used and the reasoning that prompted each step during this process in the work discussed here for the IA-32 Instruction set. We believe a similar approach needs to be adopted for a successful implementation of a machine learning classifier with other instruction sets.

Our first attempt was to use only the mnemonic of the disassembled instruction to see if that provided enough information for the accurate identification of invalid instructions in a disassembly. However, only the unique mnemonics of all positive and negative examples from the training set indicated to the learning algorithm that there was no pattern to the invalid instructions. The decision tree that was built by the algorithm was trivially classifying all instructions as legal.

We then looked at all the parts of the instruction that may be interesting for the purposes of classification. Looking at the description of the instruction structure from Intel's reference manual we were able to obtain the significant parts of any instruction. We picked the operand types, base and index registers, and the displacement and scale values, in addition to the instruction mnemonic and prefix, to represent the disassembly.

We had to then choose the encoding for each of these properties. In the IA-32 architecture, register operands were usually described by the class of the register (general purpose, segment register, control register, mmx, etc.) that could be used and the size of the register (8-bit, 16-bit, 32-bit). It did not matter which exact register was being used, as long as it belonged to the allowed class of registers. For example, the opcode 11 is the Add with Carry instruction. It is followed by additional byte(s) that specify a 16 bit general purpose register which could be any of AX, CX, DX, BX, SP, BP, SI or DI and a second operand which is either a 16 bit general purpose register or memory location. So, as long as the register is of the right type and size, it doesn't matter which one is being used. It seemed that providing the exact register was therefore information in too much detail. Intuitively interchanging these registers should not change the correctness of any disassembly. Since we are working towards building a classifier that needs to distinguish between correct and incorrect disassembly we decided to abstract away exact register names to a higher level and encode registers with just the type and the size of the registers.

Next, we chose to represent the displacement and scale values with the number of bits that was needed to represent them, since we wanted to encode not the actual value, but rather the smallness or largeness of these values, which usually seemed to be a good indicator of the validity of instructions. The actual value would again be "too concrete" to be useful for our classification purposes.

The decision tree constructed with these features did not do well on the test data. The false legal rate was extremely high (averaging around 86%) even though the false illegal rate was desirably low (0.08%). Looking into the misclassified instructions, the types of instructions that stood out had some properties that we had not captured in our feature vector. A few misclassified instructions had repeat prefixes, e.g., 'repne push eax, 'repne mov edx, esi. The IA-32 instruction set allows this prefix only with string manipulation instructions and input/ output instructions; their use with other instructions is illegal. So we decided to include a repeat-prefix feature with the values rep, repne, repz or none.

The largest number of the misclassified instructions, however, were jumps and branches. The reason for this turned out to be that many of the misclassified control transfer instructions contained an invalid target address. The validity of the target address was an abstract concept that we as humans were able to tell by looking at the address after we knew the address range of the binary, but this concept was not being captured in our feature vector. It was intuitive that the learning algorithm must use this information to create its decision tree classifier. We needed to formulate a feature that captured this notion.

The actual target address for a control transfer instruction could vary considerably from one executable file to the next; it was not the value that could be used to make the decision in an arbitrary binary. We decided to compute the validity of a target address and that would have values: VALID and INVALID. For this, we used the section header table of the executable file to obtain the virtual addresses where each section starts and ends, and use this to compute the virtual memory size of that section. We say that an immediate address N is a valid address in a given program if there is a section S, occupying the virtual address range [lo, hi], such that lo $\leq$ N $\leq$ hi. However, this additional feature did not improve the false legal rate very much.

On closer analysis, we realized that this was because of the structure of the bytes that encoded the jump and conditional jump instructions. Many of these instructions used PC-relative addressing modes with a single-byte relative displacement. These types of control transfer instructions had single byte opcodes that occurred commonly and since the target was within 128 bytes from the next instruction, this almost always fell within the current section. Therefore most of these incorrectly disassembled instructions had valid target addresses. But what stood out in many such cases was that there was either a clearly invalid instruction present at the target or none at all as in the example in Figure 4.1

In the example, the correct disassemblies are marked with an A (denoting Actual instruction address). Examining the instruction disassembled at address 0x080480d2 through the features that we picked earlier does not show any indication that it may be invalid. The branch instruction is a common one and the target address is well within the current section. But no instruction was disassembled at the target and this fact about the disassembly speaks volumes about its incorrectness.

Actual	**Address**	**Instruction**
	0x080480d2	je **80480da**
	0x080480d4	or edx,[ebp+0xffffff89]
	0x080480d7	in 0x53,eax
A	**0x080480d9**	call 80480de
A	**0x080480de**	pop ebx
A	0x080480df	add 0x89ece,ebx
A	0x080480e5	push edx

Figure 4.1: A disassembly with a branch whose target is not disassembled

Intuitively, the “weirdness” of the target instruction of the control transfer contributed to the “weirdness” of the control transfer instruction itself. We concluded that jumps and branches need to be evaluated along with their target instructions. We therefore appended the feature vector of the target instruction (if disassembled) along with that of the jump instruction to make a bi-gram. If the target was not found in the disassembly we used a dummy feature vector in its place. The decision tree constructed for such jumps alone gave promising results.

The next step was to apply the same intuition and add context to branches with immediate targets. We did this in two ways. First, we tried a single tri-gram containing (1) the branch instruction, (2) the fall through instruction followed by (3) the target instruction. In the above example, this would produce the tri-gram with instructions starting at 0x080480d2, 0x080480d4 and then a dummy instruction for the target. Second, we tried constructing two bi-grams from the branch instruction: (a) the branch instruction followed by its fall through instruction, and (b) the branch instruction followed by its target instruction. From our example the following two bi-grams would be produced by this approach: 0x080480d2, 0x080480d4 and 0x080480d2 and a dummy instruction for address 0x080480da. The second approach with two bi-grams produced better results. This also made it feasible for us to construct a single decision tree if we made bi-grams for all instructions, instead of having two separate classifiers for the uni-grams and the bi-grams.

We then noticed that the problem with invalid address references turned out to not be peculiar to control transfer instructions; other instructions had similar problems as well. This is illustrated by the following two incorrectly disassembled instructions:

or edx, [0x5d8bf689] // edx = content of address 0x5d8bf689

mov [ebx+0xcf4c01d], ecx // content of address ebx+0xcf4c01d = ecx

The first of these references an address, 0x5d8bf689, that does not occur in any of the sections in the executable file being processed; in the second, the displacement 0xcf4c01d is larger than the size of every section in the executable.

Examining these misclassified instructions suggested a possible check for the validity of any memory reference. In the compiler world, it is reasonable to assume that address computations do not cross over from one section into another. This assumption arises out of the way linkers work. The object module generated by a compiler from a source module typically consists of several code and data sections, e.g., the code section, the constant data section, the zero-initialized data section, etc. The linker combines a number of such object modules into an executable program: in the process, it puts all the sections in their final order and location. The sections of the same type coming from different object modules are typically combined into a single region of that type in the final executable. In general, when generating an object module from a source module, a compiler has no information about other object modules, e.g., their number, size, or the order in which they will be linked together, so it cannot make any assumptions about the eventual locations of these regions in the final executable. As a result, because the distance between the two regions of memory is not known at compile time, the code generated by a compiler for address computations cannot use a pointer to a particular region of memory to obtain an address pointing to some other region of memory. In other words, an address obtained by doing address arithmetic starting with a pointer to a particular region of memory can be safely assumed to fall within that same region of memory.

As an example, consider the instructions

```
mov [ebx+0xfffed495], ecx
mov [ebx+0xcf4c01d], ecx
```

In this case, if register ebx is pointing into some section, say .data, then the assumption is that the address computed using an offset off ebx also points into the same section. This means that the offset 0xfffed495 (= 76651) must be no bigger than the size of whatever section ebx is pointing into. Thus, we can give the following rule for a legal disassembly: A memory reference N(reg) is legal only if (a) N, treated as an unsigned value, is a valid address; or (b) N is a valid offset, i.e. there is a section S with size M such that $|N| \leq M$. (Note that this definition does not mean that N is in fact being used as an address or as an offset, but rather that, if N were to be used as an address/offset, it would be valid given the address ranges for the memory regions of the program.

Adding the two extra features (prefix and address validity) brought down the false legal percentage of uni-grams down to about 65%, and bi-grams down to 9%. False-illegals were now upto 0.29% for uni-grams and 2.08% for bi-grams.

On scrutinizing the false-legals again, we saw a number of instructions that had invalid addresses but which were being classified as valid, as were some control transfers which did not have their targets disassembled at all. This was because the decision tree was making decisions for some cases even without examining the address validity feature, or the features of the target instruction in the case of the control transfer instructions. Since we know that such instructions are definitely suspect without the decision tree needing to tell us that, we went ahead with implementing a two-pass classifier, where the first pass would classify the following as invalid (1) those with an invalid address in any operand and (2) those with dummy instructions. The rest of the instructions were passed to the decision tree classifier. Bi-grams passed through this two pass classifier now gave 7% false-legals and 2.1% false-illegals.

Extending the concept of bi-grams to tri-grams, following the control flow of the instructions, gave even better results with around 3.5% false-legals and 2% false-illegals. Section 4.2 formally defines an n-gram.

4.2 N-grams

A single instruction, by itself, often does not contain enough surrounding context to allow us to make an accurate determination of whether the disassembly is correct. This is partly due to the nature of the Intel IA-32 ISA, which contains a lot of short (1-byte and 2-byte) instructions and is very densely encoded, i.e., most byte sequences decode to legal instructions. Because of this, a disassembly error may not immediately produce an instruction that is recognizably erroneous. Also, some instructions typically never appear after others. One such is a push-pop sequence which could be substituted with a single move instruction. To deal with this, we consider sequences of n instruction's concatenated feature vectors instead of looking at them individually.

The obvious approach to constructing n-grams would be to consider groups of n textually adjacent instructions in the disassembly. This does not work well

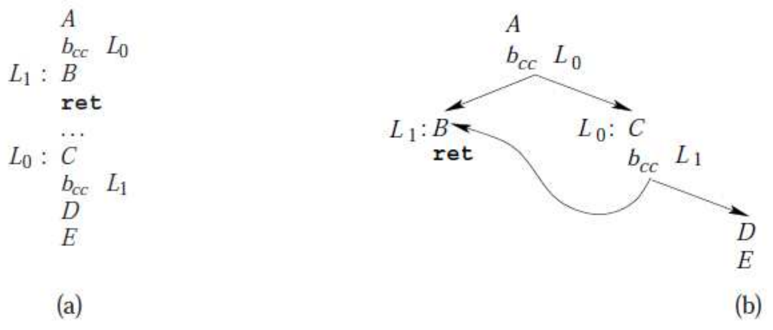


Figure 4.2: An example instruction sequence and its control flow

for control transfers, however: as an example, consider the following instruction sequence, where the address of instruction I1 is different from ℓ:

I_0 : jmp ℓ

I_1 : . . .

In this case, the 2-gram $<I_0, I_1>$ does not correspond to any instruction sequence that would actually be encountered during execution; indeed, the memory locations following I0 could contain data or garbage, i.e., not contain code at all. Thus, simply considering adjacent pairs of instructions is not enough. We therefore take a different approach. Let succs(I) denote the set of all possible control-flow successors of an instruction I, i.e, the set of instructions that could be executed immediately after I; we treat indirect control transfers (including the ret instruction, which returns from a function call) specially: since their possible successors are not known, we define succs(I) $= \emptyset$ for such instructions (this is the cause for the previously mentioned incomplete CFG).

Let $\mathrm{ng}_n(\mathrm{I})$ denote the set of all n-grams starting at instruction I, for $n \geq 1$.

This is defined as follows, with $\circ$ denoting concatenation of sequences:

$$\mathsf{ng}_n(I) = \begin{cases} \{I\} & \text{if } n = 1; \\ \{I\} & \text{if } \mathsf{succs}(I) = \emptyset; \\ \displaystyle\bigcup_{J \in \mathsf{succs}(I)} \{I \circ I' \mid I' \in \mathsf{ng}_{n-1}(J)\} & \text{otherwise.} \end{cases}$$

As an example, consider the instruction sequence shown in Figure 4.2(a), where an instruction b$_c c$ L denotes a conditional branch to the location L. The control-flow structure of this code snippet is given by the digraph in Figure fig:ngrams(b). Suppose that we are computing 5-grams for this code fragment. This can be seen as the set of paths of length 5 in this graph, except when a path is truncated due to an indirect jump. In this case, therefore, we have

$$\begin{aligned} \mathsf{ng}_5(A) = \{ & \langle A,\ b_{cc}\ L_0,\ B,\ \texttt{ret}\rangle, \\ & \langle A,\ b_{cc}\ L_0,\ C,\ b_{cc}\ L_1,\ B\rangle \\ & \langle A,\ b_{cc}\ L_0,\ C,\ b_{cc}\ L_1,\ D\rangle\} \end{aligned}$$

We use notation φ(I) to denote the feature vector for an instruction I. This lifts to n-grams and sets of n-grams in the natural way. The feature vector of an n-gram is obtained by concatenating the feature vectors of its constituent instructions:

$$\varphi(< I_1, ..., I_n >) = \varphi(I_1) \circ ... \circ \varphi(I_n)$$

Given a set of n-grams S, j(S) denotes the set of feature vectors for the elements of S:

$$\varphi(S) = \{\varphi(\alpha) | \alpha \in S\}\ .$$

4.3 Testing on malware samples

Once we got good results for benchmark tests, we started doing our tests on malware executables. For malware binaries, the process of obtaining the ground truth was through executing them and obtaining traces. We used Olly for this purpose. The resulting traces gave us the list of actual addresses at which the execution occurred. We got the offset addresses from this set. Also, with the knowledge of the parts of the binary that were self modified before execution, those packed/ encrypted bytes were also considered to be invalid bytes, if they weren't executed prior to modification.

Though this obviously did not give us the information about the entire binary, we decided to work on the information that we were able to gather and not evaluate the decision tree based on the parts of the binary that we had no knowledge about.

4.3.1 The address problem

Immediately after we tested the Decision tree classifier on the malware samples that we had, we saw plenty of false illegals. The first of the two biggest issues was because the exact number of bytes that constituted the address in windows PE files were not the same as those usually encountered in our training data which was built from gcc compiled binaries for the linux environment. This was not a problem in instructions where the address was a jump target because we were already abstracting away the number of bytes of this immediate address into the address validity field. The problem came up when addresses, as immediate values were loaded into a register and then used as an indirect address in as in the following code sample:

A 0x0041200a: mov **esi, 0x411c2c** // esi = 0x411c2c

. . .

A 0x00412019: mov dl, [**esi**] // dl = contents of address pointed to by esi

In these cases, the decision tree was making the classification based on the size of the immediate value, and the size was not one that occurred commonly in valid instructions in our training data because of the above mentioned reason.

One simple fix for this was including the address validity field for all instructions that had immediate values. Because our definition of a “valid memory address” of any N does not mean that N is in fact being used as an address or offset, but rather that, if N were to be used as an address/ offset, it would be valid given the address ranges of the program. This caused the number of misclassified instructions to go down, but only a little.

4.3.2 The 8 bit register problem

The other issue we faced with the malware samples was broader and we have been unable to deal with it sufficiently after some attempts. The issue is with the way registers are used differently in malware than in compiler generated training files.

Many of the falsely classified legal instructions frequently used 8 bit and 16 bit registers. One reason why the misclassification was occurring was that most, if not all of the appearances of these registers in the training set were in the illegal examples. This was again because the training data was created from gcc compiled binaries. Being a standard compiler, gcc hardly made use of the sub-registers for computation and therefore hardly any of the correct examples in the training set had these registers being used. On the other hand, there were plenty of these in the incorrect examples. Malware writers, on the other hand, manipulated parts of a 32 bit register for a variety of purposes - for bit masking, for obfuscating arithmetic operations, for decryption of encoded/ packed data or instruction bytes, or as a part of a bunch of dead instructions introduced only to confuse the reverse engineer.

But was there any property that needed to be true for any code written to achieve some desired behaviour? One answer to this question was that for predictable code behaviour, only initialized values are used. Thus, if a register was being used in an instruction, it probably did not matter what its size was if it was initialized before that usage.

In standard compiler terminology, a register is defined when an instruction assigns a value to it. A use of a register, on the other hand, is when an instruction uses the value of the register as one of its source operands. For an instruction to have predictable outcome along any execution path, its source registers need to be defined along all paths from the entry point to it. Usually, compilers have the complete control flow graph, since it is generating the graph from the source, it knows all the possible control flow edges that can be taken. In our case, however, we need to build the control flow graph from the possibly incorrect disassembly that we have. The presence of indirect control transfer instructions that include calls, jumps and branches with indirect target addresses, return instructions (especially when they are returns from branch functions) make the building of the control flow graph a non trivial problem. Since we could not expect to have a complete CFG, we decided to go with a less stringent condition for legal register usage. Instead of requiring that the register be defined along every path prior to its usage, we chose to consider the usage "correct" if there was some path in our incomplete CFG that defined it. We decided to mark all instructions that were not using registers "correctly" as incorrect, in the first pass of our classifier. But this did not work out as well as expected because of the incomplete control flow information.

The construction of the control flow graph was from the information present in the disassembly. Basic blocks, sequences of instructions that had only one entry point and one exit point were first constructed as follows. The set of first instructions, also called Leaders, of each basic block are obtained by adding in the following instructions (1) the entry point (2) targets of control flow instructions, if it is a direct address (3) instructions following any control transfer instruction. For each leader thus identified, its basic block is the set of instructions starting with it and going upto, but not including the next leader. The CFG was this set of basic blocks with the control flow edges between them.

It is to be noted that since we did not have all the targets of all the control transfer instructions, not all leaders could be identified and not all control flow edges were known. Many basic blocks did not have predecessors and some had incomplete predecessor lists. We found that looking for whether a register that was being used in an instruction had been defined previously, along the predecessor list in the resulting CFG, did not always find the definitions. Classifying an instruction as invalid because it used a seemingly uninitialized register caused the following types of correct instructions to be misclassified by the learned decision tree:

1. When the instruction is a part of a basic block that did not have any predecessors or if the predecessor list was incomplete because of unconditional jumps

2. When the instruction is dead and was added to the code for the sake of obfuscation. Such instructions did not need to use defined registers (as long as they made sure that they did not cause segmentation faults or exceptions)

Thus, the false illegals percentage only increased for the malware that we were using for evaluation.

Address	**Instruction**
0x0041635b	mov **dl**, 0x80
0x0041635d	mov **al**, [esi]
0x0041635f	inc esi
0x00416360	mov [edi], **al**
0x00416362	inc edi
0x00416363	add **dl** , **dl**
0x00416365	jnz 0x41636c
0x00416367	mov **dl** , [esi]
0x00416369	inc esi
0x0041636a	adc **dl**, **dl**

Figure 4.3: A disassembly with coherent use of 8-bit registers

One other thing came to light from observing the behaviour of the malware. In one case, the 8 bit register that was being used was being used repeatedly in the surrounding instructions. This usage pattern for 8 or 16 bit registers seemed right because of the following reason - using 8 or 16 bit registers may save space, and when space is being saved, intuition suggested that the same register be used repeatedly, instead of spreading the usage over many different registers, which would in a way, defeat the purpose of saving space.

Once we decided to try to incorporate this as a feature, we found that there were many ways that we could encode it. The information that we had already was the structure of the ngram, and we chose to codify coherence of the registers used and defined in a particular instruction if it was used or defined in another instruction within that ngram or any of the ngrams that had at least one instruction in common with the one being studied. For instance, in the following correct disassembly, al and dl are used over and over again: But this approach served to help only with those instructions that were used so that space was saved. In other cases, where instructions were thrown in for obfuscation, or a specific part of a register was modified by an 8 or a 16 bit bit-mask, for, say, unpacking, the decision tree still made plenty of mistakes, classifying such legal disassembly as illegal.

4.4 Feature Set

The final feature vector we used for each instruction was made up of its operation mnemonic, the repeat prefixes, the coherence of the registers being used and defined in this instrcution together with a vector of features for each of its operands. The coherence feature took on the values Coherent or Incoherent based on whether the registers in question were being used or defined in other n-grams that had at least one instruction in common with the n-gram that was being constructed.

The features we consider for the operands are the following:

– operand type, e.g., memory address; register; segment:offset pointer; immediate address for a branch instruction; etc.

– address validity, indicating whether an immediate value specified as (part of) the operand could refer to a legal address. This feature takes on one of the values Valid, Invalid or Not Applicable, depending on the operand type:

1. A memory operand containing a constant N. If the operand does not contain a base register and N is a valid address, then this feature has the value Valid, else it is Invalid. If the operand contains a base register and N is either a valid address or a valid offset then this feature is Valid, else it is Invalid.

2. An immediate operand N. For jump and call instructions, this feature is mapped to Valid if N is a valid address, Invalid otherwise. For other

instructions, this feature is mapped to Valid if N is a valid address, Not Applicable otherwise.

3. In all other cases, this feature takes on the value Not Applicable.

– displacement and/or scale values: in order to distinguish between “large” and “small” constants, we map a constant x to the number of bits in x, or to a special value indicating “ignore” if the address validity field is either Valid or Invalid.

Note that not all operands may have all of these features, e.g., an immediate operand will not have a “register” feature.

CHAPTER 5

Experimental Evaluation

5.1 Decision tree construction

To reduce total training time, we tried to avoid using training programs that were very similar to each other. To this end, we used the results of studies by Phansalkar et al. (Phansalkar et al., 2005), who examined programs from the SPEC benchmark suite and grouped them into clusters of similar programs. We used, as far as possible, a representative program from each of the clusters they identified. The training programs we used consisted of the following: applu, bzip2, fpppp, gcc, hydro2d, mcf, parser, perl, swim, and turb3d. Of these, five (bzip2, gcc, mcf, parser, and perl) are integer benchmarks, the remaining five are floating-point benchmarks. Each program was compiled at four different optimization levels, -O0, . . . , -O3, using gcc version 3.4.4, with additional command-line options to produce statically-linked binaries containing symbol table and relocation information. The resulting disassemblies contain a total of 4,377,929 correctly disassembled instructions and 10,685,094 instructions disassembled from code addresses that do not correspond to actual instruction addresses. We used the C4.5 open-source decision tree package (Quinlan, 1993)to construct our decision tree, with a minor modification to classify unseen patterns as invalid. The reason for this modification is that, intuitively, patterns that are not encountered in our extensive training set are likely to be erroneous disassemblies.

5.2 Test Inputs

5.2.1 Benchmark tests

To evaluate the accuracy of our classifier, we evaluated it on disassemblies obtained from binaries that had been deliberately obfuscated in order to introduce disassembly errors(Linn and Debray, 2003). We used as our test inputs a collection of ten programs from the SPECint-2000 benchmark suite (bzip2, crafty, gap, gzip, mcf, parser, perlbmk, vortex, and vpr), obfuscated using our anti-disassembly binary obfuscation tool. These were compiled using gcc version 3.4.4 at optimization level -O3, with additional command-line options to produce statically-linked binaries containing symbol table and relocation information (the obfuscation tool requires this to correctly update addresses after modifying the code). For each input binary P, our obfuscation tool also wrote out the set InstAddrs(P) of the addresses of the "actual" instruction in the code; this information is used to evaluate the accuracy of the decision-tree-based classifier.

These binaries were disassembled using the GNU objdump utility, and the resulting disassemblies were evaluated using our decision tree. (Oobjdump which uses a straightforward linear sweep disassembly algorithm is not a particularly good disassembler. This is not an issue here because we are interested in evaluating the accuracy of the decision-tree-based classifier, not that of the disassembler.) We extracted a set of n-grams from each disassembly, for different values of n; to facilitate evaluation, each ngram was additionally annotated with a comment (ignored by the decision tree) giving the addresses of the instructions in that n-gram. These n-grams were fed to the classifier, and the output of the classifier compared with the "ground truth" obtained from the InstAddrs(P) sets.

5.2.2 Malware tests

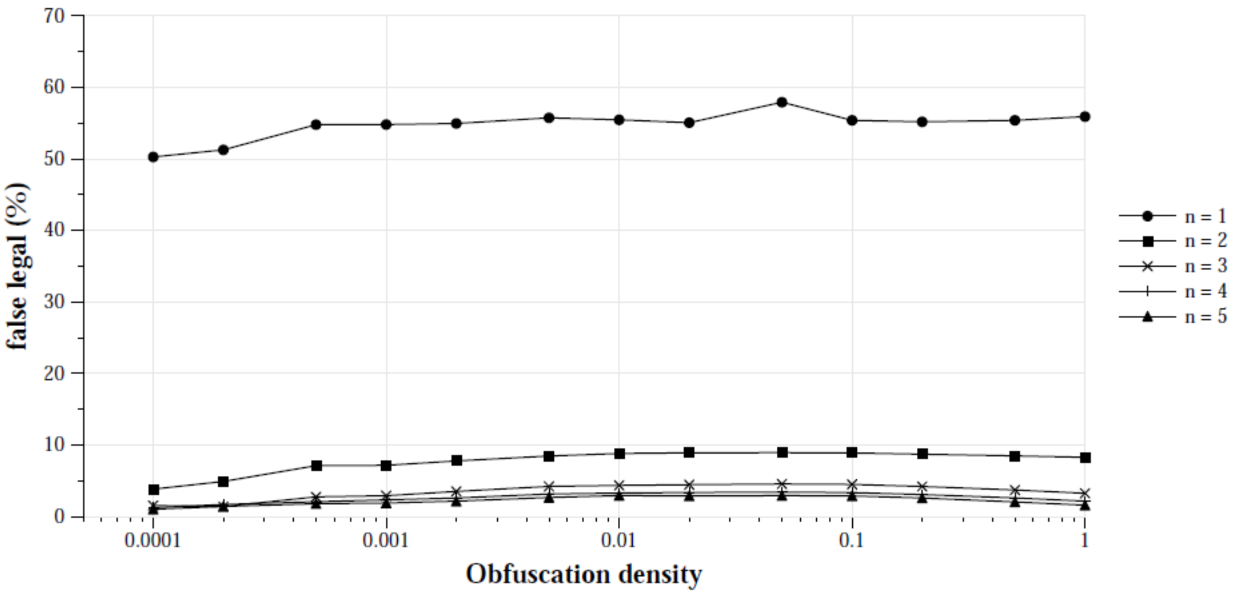
We tested our classifier on the objdump disassembly of 8 different malware binaries, Agobot.GZI.exe, Breatle.J.exe, Backdoor.Agent.YRG.exe, Backdoor.RBot.XJT.exe, Backdoor.Shark.BS.exe, Worm.Buzill.B.exe, Netsky.F.exe, Netsky.P.exe. Each of these malware executables are protected with different obfuscation tools that are available in the market. We collected execution traces for each malware binary using OllyDbg. One aspect that had to be taken care of during the execution was to see if the malware was modifying its own code, in which case, the actual address at which the execution was occurring need not be the same as the instruction disassembled at that location in the original binary. Since all the malware samples we traced were self modifying, we could not use the original executable for all the phases of the execution. (Here, a phase is defined as a set of instructions in a trace that have not been modified from the since the start of that phase. For example, lets say some amount of code is unpacked and control jumps into the newly unpacked code and no more unpacking happens. This would then make two phases, the unpacker code and then the unpacked code.)

The approach we took to collect the ground truth was to obtain the memory dump prior to each of the phases. Among the binary dump, the trace for that phase, and the knowledge about all the bytes that were modified prior to execution we had all the information we needed for our evaluation. The memory dump was statically disassembled. The instruction addresses in the trace had the InstAddrs for this phase. Those instructions that overlapped with the instructions in the trace were marked as OffsetInstrs and instructions that had bytes that were modified prior to execution in any of the following phases were added to the ModIntrs list. Both these lists together made up the address list at which there could be possibly incorrect disassemblies. Each of the phases was then evaluated by itself using its unique static disassembly. Here too, n-grams were constructed and fed to the classifier and the output compared to the "ground truth" obtained from the trace.

5.3 Classification Accuracy

5.3.1 Benchmark tests

In evaluating the accuracy of our classifier, it is important to take into account how densely packed the disassembly errors are, which depends on the extent to which the file has been obfuscated. The reason for this is that if the disassembly being evaluated contains a lot of errorsi.e., if the binary has been very heavily obfuscated everywhere, or if the disassembler happens to be very stupidthen even an imprecise classifier can produce misleadingly good results simply by blindly reporting "disassembly error" most of the time. Similarly, if the disassembly contains very few errors, a classifier can producemisleadingly good results simply by always reporting "correct disassembly." Since the code transformations introduced by the binary obfuscator are aimed specifically at throwing off the disassembly process, disassembly errors in the code tend to correlate with obfuscation points, i.e., points in the program where anti-disassembly obfuscation transformations were introduced. As a proxy for the density of disassembly errors in the input binary, therefore, we use the average density of obfuscation points in the program: given a binary with N instructions prior to obfuscation and k obfuscation points, the obfuscation density is given by k/N. A file containing no obfuscation at all has an obfuscation density of 0, while a file where obfuscation transformations are applied to every single instruction has an obfuscation density of 1. Each of the input binaries was obfuscated at 13 different obfuscating densities that range from the very sparse (104) to very dense (1.0).



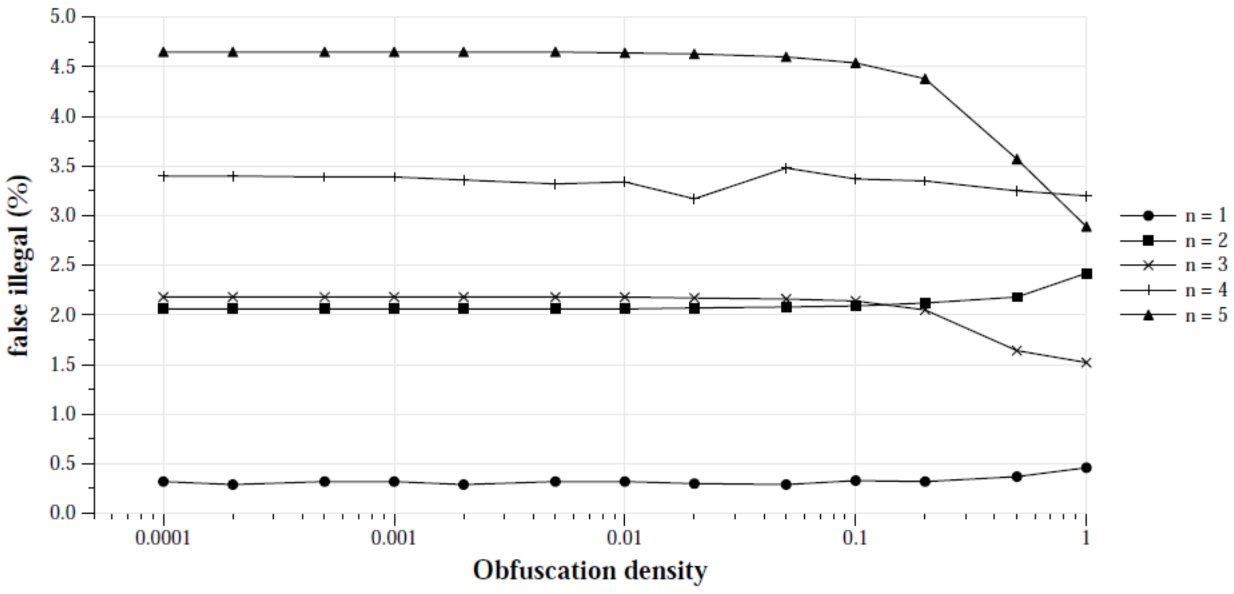


Figure 5.1: Accuracy of N-gram classification in benchmark files

Figure 5.1 shows the overall accuracy of our classifier for the different test inputs for different obfuscation densities, for n-grams with n 1,2,3,4,5. For each obfuscation density we show the average value of classification accuracy (i.e., false legals and false illegals).

It can be seen that the percentage of false legals decreases as the size of n-grams increases: it is very high for 1-grams, averaging about 50% - 58%, but drops rapidly to about 4% - 9% for 2-grams, 1.5% - 4.5% for 3-grams, 1.2% - 3.4% for 4-grams, and 1.0% - 2.9% for 5-grams. 1-grams show a great deal of variability at low obfuscation densities: e.g., at an obfuscation density of 0.0001 the false legal values for 1-grams ranges from 33% to 77%. The reason for this wide variability is that because of the low obfuscation density there are very few incorrect n-grams, which means that an erroneous classification of even a single n-gram has a large effect in terms of percentage error. As the obfuscation density increases, this variability drops quickly. The amount of variability is quite small for n-grams with n > 1. The reason for the high proportion of false legals for 1-grams is that there is very little information available about the context surrounding each disassembled instruction. As the size of n-grams increases, more and more context becomes available, leading to a steep drop in the false legal percentage. However, this gain flattens out fairly quickly: the difference between n = 3, n = 4 and n = 5 can be seen to be quite small.

The percentage of false illegals, as shown in Figure 5.1 is quite small for all values of n, ranging from 0.2% to 4.6%. Interestingly, false illegals are lowest for 1-grams, ranging from 0.2% to 0.4%; they are about 2.1% for 2-grams, about 2.07%for 3-grams, about 3.3% for 4-grams, and about 4.4% for 5-grams. The reason for the low proportion of false illegals for 1-grams is essentially the same as that for the high false legal rate for this case: only those instructions that are very obviously

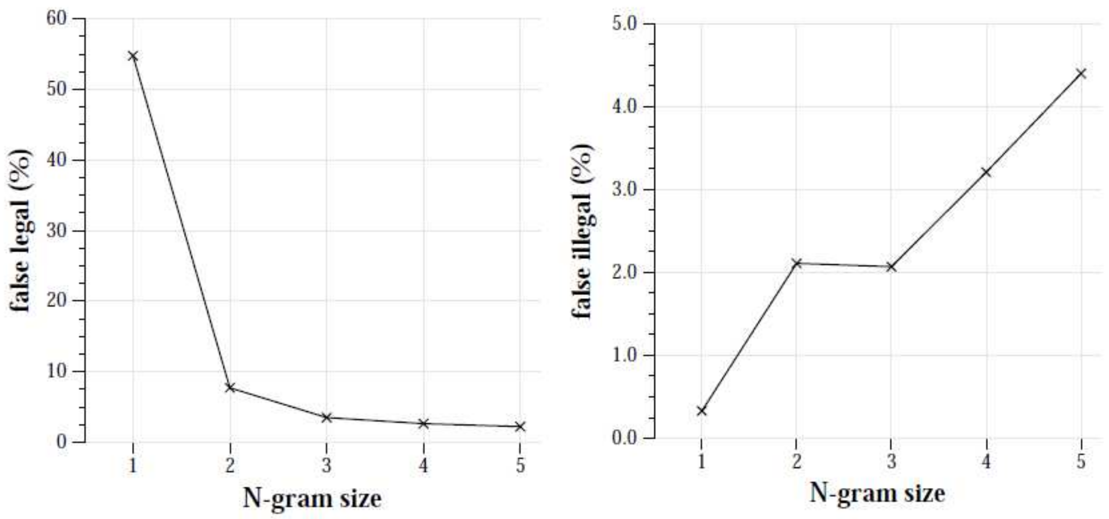


Figure 5.2: Variation of false legal and false illegal with N-gram size

incorrectly disassembled are classified as incorrect. The reason for the relatively high false illegal rate for 4-grams is that if the input contains an n-gram that does not occur in the training set, it is by default classified as "incorrect." As the value of n increases, therefore, one expects there to be more and more n-grams that occur in the input but which may not have occurred in the training set, and which are therefore falsely classified as illegal. This is borne out by Figure 5.2, which shows that the percentage of false legals drops quickly as the size of n-grams increases while the percentage of false illegals increases.

By and large, the results are quite stable: the average false legal and false illegal percentages do not vary much for different obfuscation levels (1-grams are an exception to this, but this is in some sense moot because the high level of false legals in this case limits its practical utility anyway.) This is desirable, because it suggests that decision-tree-based classifiers are not overly sensitive to the density of disassembly errors.

Overall we find that for the programs we studied, 4-grams give the best results, with false legals ranging from 1.2% to 3.4% and false illegals ranging from 3.2% to 3.3%.

5.3.2 Malware tests

Figure 5.3 and Figure 5.4 show the overall accuracy of the classifier for the different malware samples. For n-grams with $n \in \{1, 2, 3, 4\}$ we show the average value of classification accuracy among all the phases of that malware (i.e., false legals and false illegals).

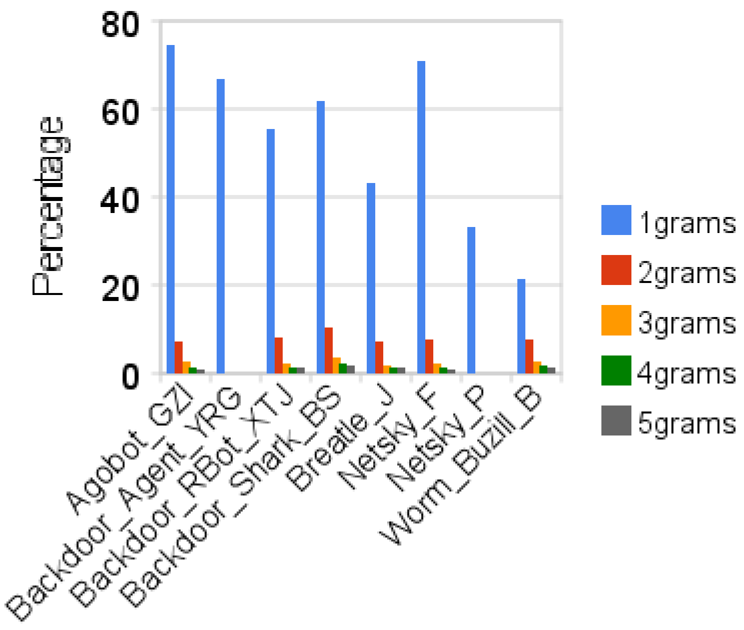


Figure 5.3: False legals in malware

False legals are very high for n=1 averaging around 53% for all the malware, but drops rapidly as the size of n increases going to 6% for 2-grams, 1.9% for 3-grams, 1.7% for 4-grams and 0.9% for 5-grams for the same reason that this happens with our banchmark files. As more context is available, the false legals go down.

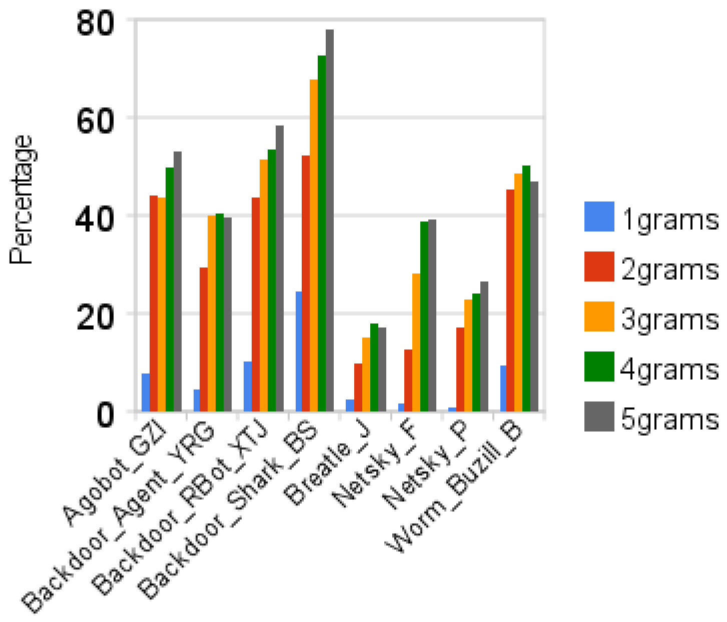


Figure 5.4: False illegals in malware

The false illegals increase rapidly as the size of the ngram increases. The reason for the high percentage of false legals in the case of malware is that the instructions are very dissimilar from the training inputs. The three major differences that we have identified as the cause the high false illegal ratio for the 2-grams and 3-grams are the following:

1. There are many uses of abnormal immediate values in correctly disassembled instructions. Such values occur frequently in the incorrect samples in the training set. An example is an instruction of the form:
   push 0xa894b25
   The value 0xa894b25 here is not a part of the address range of the binary and the size of this value is not commonly seen in our positive training set.

2. Mnemonics that were not seen in the correct samples in the training set are

used very often in the malware. Examples are lodsd (loading a double word from the address pointed to by the data segment register offset by the source index register into EAX), pushad (pushing all general purpose registers onto the stack) and stc (setting the carry flag). Some instructions such as adc (add with carry) were present in the positive training set, but in a far greater number than in the negative set.

3. There are sequences of correctly disassembled instructions in malware that are not seen in the training set like the following example:
   push ecx
   pop edi
   popad

One more point to be considered is that none of the traces we collected were too long, the average number of actual instructions executed for each trace file is 360, with just four traces having more than a 1000 unique malware instructions being executed, even though the smallest malware binary file that we considered was about 15K. This could make the percentage values of the results to appear large even though the number of inaccurate classifications were small. In our 2-grams, for instance, among the 16 malware phases whose false illegal percentage turned out to be greater than 50%, 9 of them had less than 50 2-grams that were correctly disassembled, 3 had less than 100 and the remaining four had 298, 306, 892 and 1315 correct 2-grams.

Overall, the identification of the incorrectly disassembled instructions is quite accurate in the case of Malware, with 3-grams giving the best results with false legals ranging from 2% to 11% and false illegals ranging from 10% to 81% . Even though the percentage of false positives is high, it is important for our purposes that we do not miss the incorrect disassemblies and our classifer acts as desired in these situations.

CHAPTER 6

Future work and Conclusion

6.1 Multi-stage Classifiers

The two-stage classifier that we built is seen to be insufficient to handle the special types of instruction sequences that we see in malware binaries. One idea that may be well worth trying is to collect enough data from malware binaries and train a decision tree using this as the training set. The resulting decision tree will be specialized for obfuscated and handwritten code because it will have examples that do not typically appear in gcc compiled binaries, but in malware binaries. These two classifiers can then be polled and the weighted result can be obtained.

Other machine learning techniques could also be applied to the same problem. One such, the AdaBoost algorithm (Freund and Schapire, 1996), is known to consistently outperform other classifiers.

6.2 A Hybrid Static/ Dynamic Disassembler

The bigger problem that we are trying to solve is to try to obtain a correct disassembly of arbitrary binaries. The solution proposed in this thesis is just the first step toward this goal, an identification of possible errors in the disassembly. Having a small number of false legals is necessary for a dynamic analysis that is based on this identification to work well. We have shown that this is the case even for our malware samples.

Once this is list of potentially incorrect disassemblies is obtained, the next step will be to try and examine these portions of the disassembly more closely to get the right control flow. One way that this can be done is by dynamic multi-path analysis(Moser et al., 2007), where execution is forced through alternate paths in a controlled environment. Our predictions about the correctness of the static disassembly, using the proposed approach, can be used to prioritize the sections of the code that need to be examined.

6.3 Conclusions

Disassembly of executables is an important component of reverse engineering binary code to understand its semantic meaning. Disassembly errors can lead to errors in higher level semantic analyses based on the disassembly and can also cause some code to be missed from analysis. Unfortunately, and especially on the widely used Intel IA-32 architecture, disassembly errors very often do not result in obvious problems such as illegal opcodes or instructions, but produce other legal instructions that are not always easily distinguished from those in a correct disassembly. This paper presents a machine learning approach to static identification of disassembly errors. Experimental results from a prototype implementation, using disassemblies obtained from a variety of obfuscated executables, indicate that the approach can accurately classify over 75% of the instructions even for heavily obfuscated benchmark files. Among malware samples, more than 95% of the known incorrect disassemblies are identified, with a false positive ratio of about 50%.

REFERENCES

(2009). ASProtect software. http://www.aspack.com/asprotect.aspx.

Balakrishnan, G. and T. Reps (2004). Analyzing memory accesses in x86 executables. In *Proc. 13th. International Conference on Compiler Construction*, pp. 5–23.

Balakrishnan, G., T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. Yong, C.-H. Chen, and T. Teitelbaum (2005). Model checking x86 executables with codesurfer/x86 and WPDS++. In *Proc. Computer-Aided Verification.*

Christodorescu, M. and S. Jha (2003). Static analysis of executables to detect malicious patterns. In *Proc. 12th Usenix Security Symposium*, pp. 169–186.

Christodorescu, M., S. Jha, S. A. Seshia, D. Song, and R. E. Bryant (2005). Semantics-aware malware detection. In *Proc. Usenix Security '05.* To appear.

Freund, Y. and R. E. Schapire (1996). Experiments with a new boosting algorithm.

Intel Corp. (2008). *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference.* Intel Corp.

Kruegel, C., W. Robertson, and G. Vigna (2004). Detecting kernel-level rootkits through binary analysis. In *Proc. 20th Annual Computer Security Applications Conference (ACSAC).*

Linn, C. and S. Debray (2003). Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th. ACM Conference on Computer and Communications Security (CCS 2003)*, pp. 290–299.

Moser, A., C. Kruegel, and E. Kirda (2007). Exploring multiple execution paths for malware analysis. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pp. 231–245.

Objdump (2005). *GNU Manuals Online.* GNU Project—Free Software Foundation. `http://www.gnu.org/manual/binutils-2.10.1/html_chapter/binutils_4.html`.

Phansalkar, A., A. Joshi, L. Eeckhout, and L. K. John (2005). Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *ISPASS '05:*

Proc. IEEE International Symposium on Performance Analysis of Systems and Software, 2005, pp. 10–20.

Popov, I., S. K. Debray, and G. R. Andrews (2007). Binary obfuscation using signals. In *Proc. Usenix Security 2007*, pp. 275–290.

Prasad, M. and T. Chiueh (2003). A binary rewriting defense against stack based buffer overflow attacks. In *Proc. USENIX Technical Conference*.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning* **1**, 81–106.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Schwarz, B., S. K. Debray, and G. R. Andrews (2002). Disassembly of executable code revisited. In *Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE)*, pp. 45–54.

Singh, P. K., M. Mohammed, and A. Lakhotia (2003). Using static analysis and verification for analyzing virus and worm programs. In *Proc. 2nd. European Conference on Information Warfare*.

Xu, Z., B. P. Miller, and T. Reps (2000). Safety checking of machine code. In *Proc. ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pp. 70–82.

Yuschuk, O. (2009). Ollydbg. `http://www.ollydbg.de/`.