Static Detection of Disassembly Errors

Nithya Krishnamoorthy Saumya Debray Keith Fligg†
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA
Email: nithyak@cs.arizona.edu, debray@cs.arizona.edu, keith.fligg@pnl.gov

***Abstract*—Static disassembly is a crucial first step in reverse engineering executable files, and there is a considerable body of work in reverse-engineering of binaries, as well as areas such as semantics-based security analysis, that assumes that the input executable has been correctly disassembled. However, disassembly errors, e.g., arising from binary obfuscations, can render this assumption invalid. This work describes a machine-learning-based approach, using decision trees, for statically identifying possible errors in a static disassembly; such potential errors may then be examined more closely, e.g., using dynamic analyses. Experimental results using a variety of input executables indicate that our approach performs well, correctly identifying most disassembly errors with relatively few false positives.**

***Keywords*-disassembly; reverse engineering; binary analysis; machine learning;**

I. Introduction

Static disassembly, which recovers an assembly instruction sequence from an executable file, is a crucial first step in reverse engineering executable files. It therefore finds important applications in computer security: researchers confronted with a suspicious executable typically disassemble its code in order to determine its internal logic, and there is an extensive body of security research that assumes that the input file has been correctly disassembled [4], [8], [21].

This assumption may not always hold in practice, however. Disassembly errors can arise due to a variety of reasons, and may be accidental or deliberate. An accidental disassembly error occurs when a disassembler somehow misinterprets some part of an executable file; very often, this happens when legitimate data, such as a jump table, happens to be embedded in the code stream and is misidentified as code. Disassembly errors may also be induced by deliberate binary-level obfuscations aimed at protecting the code against reverse engineering; in earlier work we have shown that it is surprisingly easy to fool even state-of-the-art disassemblers into making errors that cause large parts of the input programs ($\sim$65% of the instructions and $\sim$85% of the functions) to be incorrectly disassembled [9], [15]. Unfortunately, it is not always easy to determine whether an executable has been correctly disassembled. This is especially true of the widely-used Intel IA-32 architecture, where a combination of variable-length instructions and high encoding density (most byte sequences decode to valid instructions) makes it difficult to identify erroneous disassemblies.

The most definitive validation of a disassembly is to execute a program and observe the instructions that get executed, but such a dynamic analysis can only provide information about a single execution path through a program. It is possible to extend dynamic analyses to explore multiple alternative execution paths [12], but in practice it may not always be practical to dynamically explore an entire program in this way, for two reasons: first, the number of distinct execution paths through a program can grow exponentially as the number of basic blocks; and second, determining the specific conditions that force execution to reach a particular code fragment may be difficult. Furthermore, multipath dynamic analysis can be misled by dynamic defenses based on memory errors [3]. This argues for a method to statically inspect a disassembly and identify regions that may contain disassembly errors. Using such an approach, for example, one might assign higher priority for dynamic analysis to regions of code whose disassembly appears suspect. This paper describes an approach that uses machine learning techniques to accomplish this. It is based on the observation that erroneous disassemblies are usually statistically quite different from correct disassemblies, and may contain, for example, instructions with uncommon opcodes or operands, nonexistent branch target addresses, etc. We use training sets of correct and incorrect disassemblies to train a decision-tree-based classifier. Experimental results indicate that with appropriate training, such classifiers can be quite accurate, and can correctly identify most disassembly errors in test files.

The remainder of this paper is organized as follows. Section II discusses background material on disassembly and decision tree models. Section III discusses the ef-

† Current address: Keith Fligg, Pacific Northwest National Laboratory, Richland, WA 99352.

fects of binary obfuscation on disassembly and shows how disassembly errors can lead to security problems. Section IV discusses decision tree models for static disassembly. Section VI gives experimental results from a prototype implementation of our ideas. Section VII discusses related work, and Section VIII concludes.

II. Background

This section discusses background material on disassembly algorithms and decision trees. It may be skipped by readers familiar with these topics.

A. Disassembly

Broadly speaking, there are two approaches to disassembly: *static* and *dynamic*, the difference between them being that the former examines the program without execution, while the latter monitors the program's execution (e.g., through a debugger or emulator) as part of the disassembly process. Static disassembly processes the entire input program all at once, while dynamic disassembly only disassembles those instructions that were executed for the particular input that was used. Moreover, with static disassembly it is easier to apply offline program analyses to reason about semantic aspects of the program under consideration. Finally, programs being disassembled statically are not able to defend themselves against reverse engineering using anti-debugging or anti-monitoring techniques [5]. For these reasons, static disassembly is a popular choice for low level reverse engineering. This paper focuses on static disassembly.

Static disassemblers typically use one of two techniques: *linear sweep* and *recursive traversal* [20]. With linear sweep, disassembly begins at the program's first executable location and proceeds sequentially, disassembling each instruction as it is encountered. This method is used by programs such as the GNU utility `objdump` [13] as well as a number of link-time optimization tools. Recursive traversal, by contrast, starts with the program's entry point, and disassembles the code sequentially until a control transfer instruction is encountered. The disassembler then recursively processes the possible control flow successors of that instruction, i.e., addresses where execution could continue. Variations on this basic approach to disassembly are used by a number of binary translation and optimization systems. A recently proposed generalization of recursive traversal is that of exhaustive disassembly [6], [7]. This approach aims to work around certain kinds of binary obfuscations by considering all possible disassemblies of each function. It examines the control transfer instructions in these alternative disassemblies to identify basic block boundaries, then uses a variety of heuristic and statistical reasoning to rule out alternatives that are unlikely or impossible.

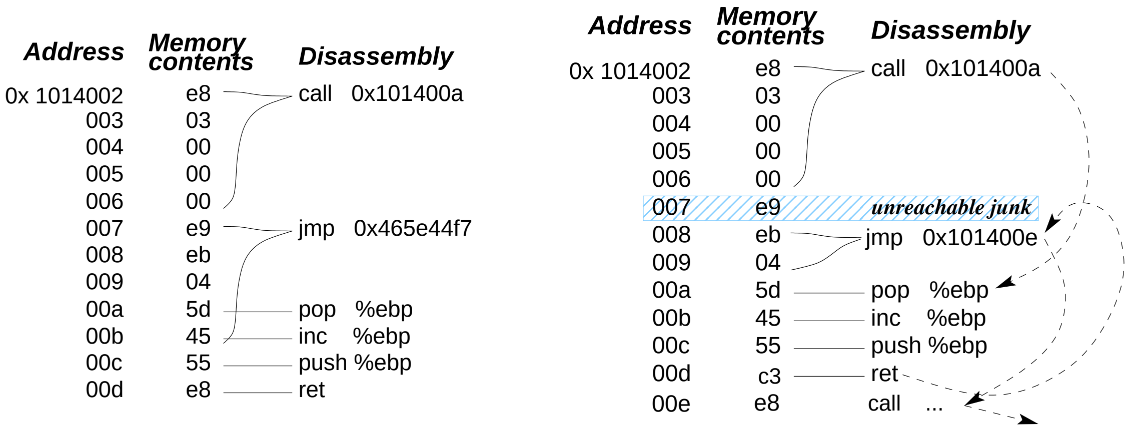
Interestingly, the disassembly process on the Intel IA-32 architecture turns out to be *self-repairing* [9]. What this means is that when a disassembly error occurs, a few instructions may be incorrectly disassembled, but the disassembly process eventually corrects itself (usually fairly quickly, within $3-5$ instructions). It then proceeds to produce a correct disassembly thereafter, until the next disassembly error is encountered.

B. Decision Trees

Decision trees are a technique for supervised machine learning, i.e., learning a function from a training data set. The training set consists of a set of pairs $(x, f(x))$ where f is the unknown function to be learned. The task of the learning process is to see some number of pairs (x, y), where y is the value of the function f at the point x, and from this "learn" f, i.e., be able to predict the value of $f(x)$ for values of x that are not in the training set. For our purposes, the training input consists of *positive examples*, i.e., pairs of the form $(x, \mathsf{correct})$ where x is a correctly-disassembled code sequence, and *negative examples*, i.e., pairs of the form $(x, \mathsf{incorrect})$ where x is an incorrectly-disassembled code sequence. The desired output is a classifier that is able to distinguish between correct and incorrect disassemblies.

In general, each item in the domain of the function f to be learned is considered to be defined by a collection of *features* together with a set of possible values for each feature. In the context of the problem of detecting disassembly errors, for example, such features might include '*opcode*', '*source addressing mode*', '*destination addressing mode*', etc., with different values for different instructions. The essential idea in decision tree construction is to examine the various features of the training data to come up with a "good" set of tests, organized in the form of a tree, that can be used to distinguish between the positive and negative examples. Ideally, we want the decision tree computed for any given training set to be as small as possible; however, it turns out that the construction of minimal decision trees, either in terms of nodes, leaves, or depth, is an NP-hard problem.

Because of this NP-hardness result, most decision tree construction algorithms use greedy heuristics; in particular, an approach based on the ID3 system of Quinlan [16], which attempts to maximize the information gain at each step, is widely used in practice. This is done using the information-theoretic notion of *entropy*. Given a collection of items $S =$



(a) static disassembly

(b) actual instruction sequence

Figure 1. An example of disassembly errors in an obfuscated binary

$\{s_1, \ldots, s_n\}$, such that $s_i \in S$ occurs with probability p_i, the entropy of S is given by

$$H(S) = -\sum_{i=1}^{n} p_i \log p_i.$$

Given a collection of training items S with features $F_1, \ldots, F_k$, the information gain for a feature F_i is given by the expected reduction in entropy resulting from splitting on this feature:

$$Gain(S, F_i) = H(S) - \sum_{v \in V_i} (|S_v|/|S|) H(S_v)$$

where V_i denotes the set of values taken on by feature F_i, and S_v is the subset of S having the value v for feature F_i [11]. In this computation, the entropy of each resulting subset is weighted by the relative size of that subset. The decision tree construction process proceeds recursively, starting with the entire training set and at each step choosing a feature that yields the maximal information gain. The recursion stops when a set of items is obtained that are all in the same category, i.e., where the function f to be learned has the same value for all of the items in the set.

III. Disassembly Errors and Binary Obfuscation

To see the potential implications of disassembly errors, we consider the static disassembly of the trojan *Win32.Microjoin.R*, whose executable was packed using the software protection tool *Aspack* [1]. Straightforward recursive disassembly produces the innocuous code sequence shown in Figure 1(a): the code seems to consist of just six instructions, which don't do much. (The recursive disassembly algorithm does not find any other code here because it uses the possible targets of control transfer instructions to discover code. In this example, the target address `0x465e44f7` is not in the address range of the binary, and there are no other control transfers to follow.)

The actual code for this part of the program is shown in Figure 1(b). It turns out that the packer uses two techniques, "branch functions" and junk-byte insertion [9], to obfuscate the binary: the target of the *call* instruction actually modifies its return address (using the pop–inc–push instruction sequence), so that the call returns to an address one byte beyond the return address passed to it by the *call* instruction. This small perturbation to the return address is enough to cause the disassembly to miss the real jump instruction, '`jmp 0x101400e`', at address 0x1014008. Similar snippets of obfuscated code follow, which eventually cause the malware payload to be unpacked and executed.

There are two points to note in this example. The first is that carefully crafted binary obfuscations can lead to disassembly errors that cause code to be missed by the disassembler, and so also by higher-level analyses that rely on this disassembly. The second is that an error in disassembly may not produce anything as obvious as an illegal opcode or instruction. Because of this, identifying potential errors in a large static disassembly can be quite difficult.

The example above has some telltale signs that the disassembly may not be correct and/or complete. The most striking of these is the jump to an address that does not exist in the code, but this is not something that can be guaranteed to occur in all erroneous disassemblies. Another potentially suspicious feature of the disassembly in Figure 1(a) is that the call instruction at the beginning seems to jump into the middle of another instruction, but it is not inconceivable that this might be the result of a clever compiler optimizing for code size. Finally, most of the code section is not disassembled in this example, which may seem suspicious. However, this again does not generalize to all disassembly errors, since in order to hide malicious content it suffices to conceal only a few key instructions.[1] The remainder of the paper discuss our approach to identifying some of these telltale signs that may indicate an error in disassembly.

IV. Decision Tree Models for Static Disassembly

A. Instruction Feature Extraction

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 on. In many cases, it turns out that what is of interest is not the actual value of some aspect of an instruction, but rather a property of that value. For example, in the instruction

```
jmp 0x465e44f7
```

in the example shown in Figure 1, what is most significant is not the actual value of the branch target address, but rather that it is too large to be a legal address. To capture such properties, when extracting feature values from instructions we abstract away from operand values that are "too concrete."

While the actual target address for a control transfer instruction may be too concrete, however, it turns out to be important to know whether a constant might specify a memory address. This can happen if either the constant is itself a valid address, or else if it specifies an offset from some other address (in the latter case, which arises for example when computing the address of a field within a heap-allocated structure, the base address is typicaly computed into a register and is not readily discernible from the code). To determine this, we use 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 a constant N is a *valid address* in a given program if there is a section S, occupying the virtual address range $[lo, hi]$, such that $lo \leq N \leq hi$; we say that N is a *valid offset* if there is a section S with size M such that $|N| \leq M$. (Note that this 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.)

[1]As an example, consider a simple decryptor loop that decrypts and executes code in a section that—as in the Aspack-protected code described above—is not marked as containing code. The decryptor can be written using as few as 8–10 instructions, which can be scattered through a program, with the code and control transfers between them masked using careful obfuscation, as above. In such a case, the amount of undisassembled memory in the code section would be quite small.

For each instruction, we use as its feature set its operation mnemonic and the repeat prefix together with a vector of features for each of its operands. In the IA-32 instruction set, a particular class of operations, e.g., push or jump, may be encoded via a number of different byte values at the binary level; these are all abstracted into a single operation mnemonic. The repeat prefixes can be used only with string and input/ output instructions and therefore can be helpful in identifying invalid disassemblies when it occurs with other instructions. 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.
– *base and/or index registers*, mapped to the size of the register (i.e., 8-bit, 16-bit, 32-bit) if a general-purpose register, or else to the type of register (segment register, control register, mmx, 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. Note also that we create equivalence classes for "similar kinds" of operands, e.g., all 32-bit general-purpose registers get mapped to a single feature value '*GPREG_32*.' We use the notation $\varphi(I)$ to denote the feature vector for an instruction I.

B. Instruction 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. To deal with this, we incorporate additional execution context into our feature vectors by considering n-grams of instructions, i.e., groups of n instructions that may be executed in sequence.

The obvious approach to constructing n-grams would be to consider groups of n adjacent instructions. This does not work well for control transfers, however: as an example, consider the following instruction sequence, where the address of instruction I_1 is different from ℓ:

$$\begin{array}{ll} I_0: & \texttt{jmp}\ \ell \\ I_1: & \ldots \end{array}$$

In this case, the 2-gram $\langle I_0, I_1\rangle$ does not correspond to any instruction sequence that would actually be encountered during execution; indeed, the memory locations following I_0 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 $\mathsf{succs}(I)$ 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 $\mathsf{succs}(I) = \emptyset$ for such instructions. Let $\mathsf{ng}_n(I)$ 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 2(a), where an instruction '$b_{cc}\quad L$' denotes a conditional branch to the location L. The control-flow structure of this code snippet is given by the digraph in Figure 2(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

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

Recall that the feature vector for an instruction I is denoted by $\varphi(I)$. 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:

$$\varphi(\langle I_1, \ldots, I_n\rangle) = \varphi(I_1) \circ \cdots \circ \varphi(I_n).$$

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

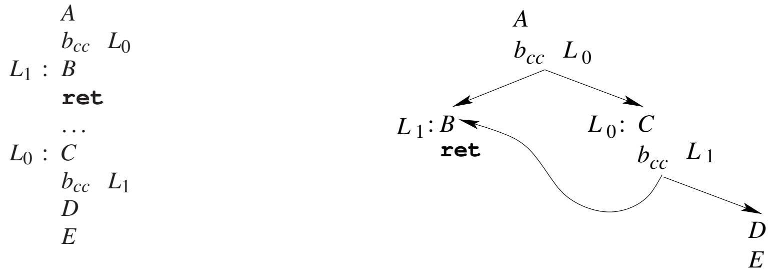
$$\varphi(S) = \{\varphi(\alpha) \mid \alpha \in S\}.$$

In the discussion that follows, we will sometimes abuse notation and write $\mathsf{ng}_n(\ell)$, where ℓ is an address, to refer to the set of n-grams starting with the instruction at address ℓ.

C. Decision Tree Construction

Once we have identified the set of features and feature values of interest, the next step is to identify appropriate sets of "correct" and "incorrect" disassemblies, extract the appropriate feature vectors for them, and use these as training inputs to construct a decision tree. This confronts us with the problem of identifying a sufficiently large set of instructions that can be guaranteed to have been correctly disassembled. This is a difficult problem in general, since given an arbitrary executable, and without any other auxiliary information, the only way we can be certain of the "correct" disassembly is to execute the program and record the instructions that are executed at different addresses. However, this would only yield a disassembly for a single execution path through the program; even using multipath execution techniques [12], we would be able to explore at best a small fraction of all of the possible execution paths through the code, so it is not clear whether we would be able to disassemble all, or even most, of the program. Furthermore, using dynamic multipath execution to explore a significant number of alternative execution paths through a large program could be quite time-consuming.

For this reason, we take a different approach. We use *gcc*-compiled binaries containing symbol table and relocation information. This turns out to be sufficient for accurate static disassembly, since the relocation information is enough to



(a) (b)

Figure 2. An example instruction sequence and its control flow structure

identify code addresses, and therefore the targets of indirect jumps and calls (the targets of direct jumps and calls are easily determined).[2] From each binary P, we extract the set of addresses $\mathsf{InstAddrs}(P)$ for all of the instructions in P using the PLTO binary rewriting system [19]. Suppose that the code section of the program P spans an interval of addresses $\mathbf{A}$. We begin by computing, for the specific value of n we are using, the set of all n-grams for P, i.e., starting at every address in $\mathbf{A}$:

$$\mathsf{NG}_n(P) = \bigcup_{a \in \mathbf{A}} \mathsf{ng}_n(a).$$

Let $\mathsf{addr}(I)$ denote the address of an instruction I. The set of "good" n-grams for P, denoted by $\mathsf{NG}_n^{good}(P)$, are those that do not contain any disassembly errors, i.e., where each instruction in the n-gram is at an address in $\mathsf{InstAddrs}(P)$:

$$\mathsf{NG}_n^{good}(P) = \{\alpha \in \mathsf{NG}_n(P) \mid \forall I \in \alpha : \mathsf{addr}(I) \in \mathsf{InstAddrs}(P)\}.$$

The set of positive training inputs is then given by the set of feature vectors for such good n-grams:

$$\mathsf{PosInputs}(P) = \varphi(\mathsf{NG}_n^{good}(P)).$$

To obtain the set of negative training inputs, we begin with the set of "bad" n-grams, i.e., those where at least one instruction in the n-gram does not correspond to an address in $\mathsf{InstAddrs}(P)$:

$$\mathsf{NG}_n^{bad}(P) = \mathsf{NG}_n(P) - \mathsf{NG}_n^{good}(P).$$

We cannot simply take the negative training inputs to be the feature vectors for these n-grams, however, because the encoding density of the Intel IA-32 instruction set can cause the feature vectors of the good and bad n-grams to overlap. Since it makes little sense to consider a feature vector to be both "good" and "bad", we remove any such overlap to obtain the negative training input:

$$\mathsf{NegInputs}(P) = \varphi(\mathsf{NG}_n^{bad}(P)) - \mathsf{PosInputs}(P).$$

Once the positive and negative training sets are obtained, we find that it is always the case that the number of positive examples are much fewer than the number of negative examples. So that the results are not skewed greatly towards the negative, we keep adding in the unique positive set to the training set make the number of positive examples greater than or equal to the number of negative examples.

V. Evaluation Metrics

We consider two metrics for evaluating decision-tree-based classifiers: 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.

In order to define these notions more formally, we have to specify what we mean when we refer to some part of the program as being correctly or incorrectly disassembled. Suppose that we know, for a program P, the set $\mathsf{InstAddrs}(P)$ of the addresses of all of the instructions in P. An n-gram α is said to be correctly disassembled if, for each instruction I in α, the address of I is in $\mathsf{InstAddrs}(P)$; otherwise α is incorrectly disassembled.

A *false legal* refers to an n-gram that is incorrectly disassembled but which is erroneously classified as correct; a *false illegal* refers to an n-gram that is correctly disassembled but which is erroneously classified as incorrect. The percentage of false legals in the disassembly is then given by the fraction of incorrectly disassembled n-grams that are false legals; the false illegal percentage is the fraction of correctly disassembled n-grams that are false illegals.

[2]The executables intended as the eventual targets for our approach will not, of course, contain such detailed information, but this is not a problem since we are using the symbol table and relocation information only to construct the training inputs.

VI. Experimental Results

To evaluate our ideas, we implemented a classifier for disassembly errors based on decision trees, as discussed above. This section reports the results of our experiments. The numbers reported were obtained on a workstation with a 2.4 GHz Intel dual-core x86-64 processor, with 4GB of RAM and 640 GB of hard disk, running Linux 2.6.21-1.3228.fc7.

A. 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.* [14], 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 [17] to construct our decision tree.

B. Test Inputs

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 [9]. 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 [9]. 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 $\mathsf{InstAddrs}(P)$ of the addresses of the "actual" instruction in the code; this information is used to evaluate the accuracy of the decision-tree-based classifier, as discussed in Section V.

These binaries were disassembled using the GNU *objdump* utility [13], and the resulting disassemblies were evaluated using our decision tree. (As discussed in Section II-A, *objdump*—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, as discussed in Section IV-B; to facilitate evaluation, each n-gram 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 $\mathsf{InstAddrs}(P)$ sets.

C. Classification Accuracy

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 errors—i.e., if the binary has been very heavily obfuscated everywhere, or if the disassembler happens to be very stupid—then 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 produce misleadingly 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 (10^{-4}) to very dense (1.0).

Figure 3 shows the overall accuracy of our classifier for the different test inputs for different obfuscation densities, for n-grams with $n \in \{1,2,3,4\}$. For each obfuscation density we show the average value of classification accuracy (i.e., false legals and false illegals) together with error bars indicating the range of values for the different programs.

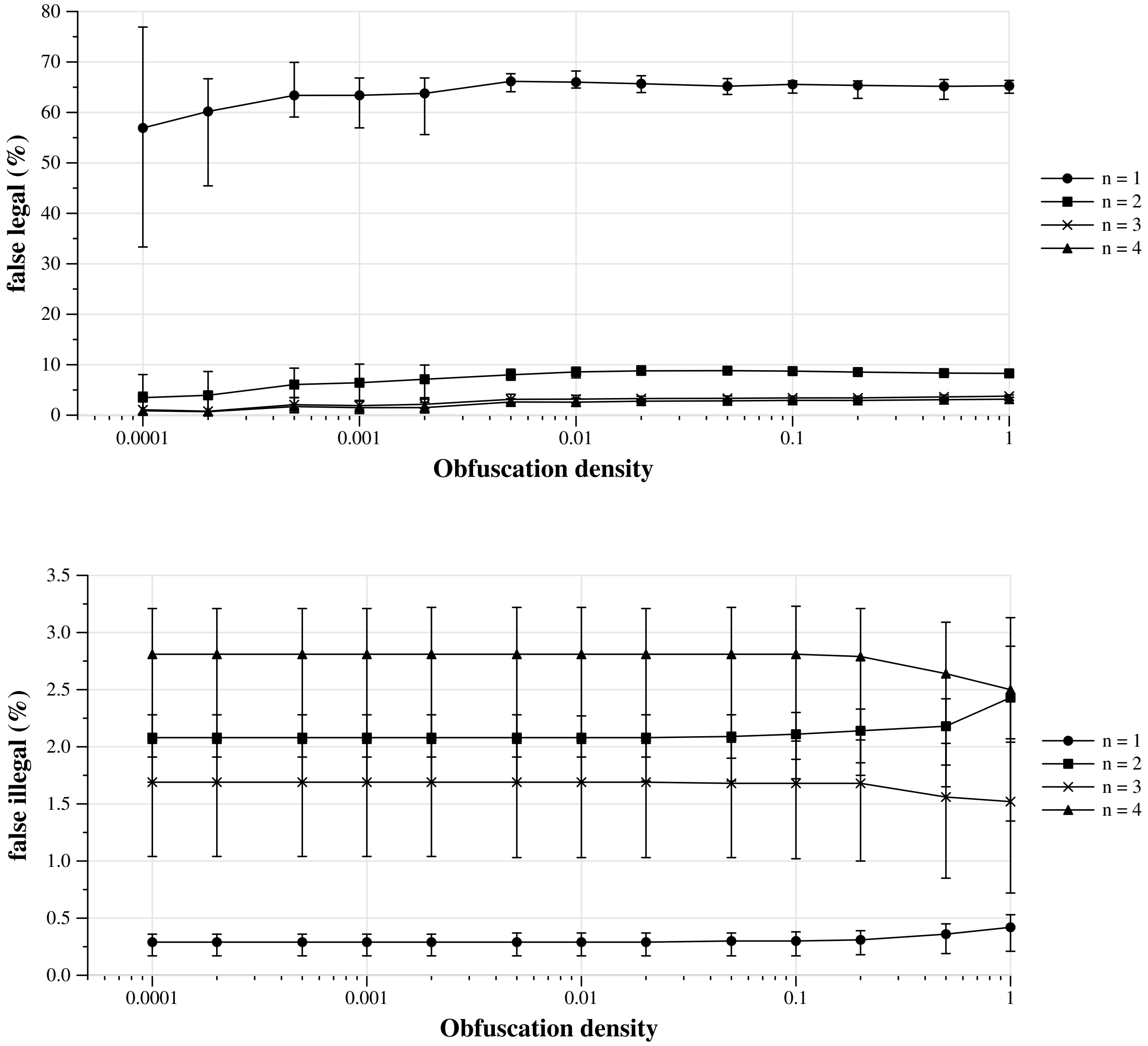


Figure 3. Accuracy of *n*-gram classification

It can be seen from Figure 3 that the percentage of false legals decreases as the size of *n*-grams increases: it is very high for 1-grams, averaging between 57% and 66%, but drops rapidly to about 3.5%–8.8% for 2-grams, 0.8%–3.8% for 3-grams, and 0.7%–3.2% for 4-grams. 1-grams also 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 $n = 3$ and $n = 4$ can be seen to be quite small.

The percentage of false illegals is quite small for all values of *n*, ranging from 0.4% to 2.8%. Interestingly, false illegals are lowest for 1-grams, ranging from 0.3% to 0.4%; they are a little over 2% for 2-grams, about 1.7% for 3-grams, and about 2.8% for 4-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 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 the false illegal percentages for $n = 4$, which are higher than for smaller values of n. (At this time we do not know why 3-grams have a smaller proportion of false illegals than 2-grams.)

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, 3-grams give the best results, with false legals ranging from 0.8% to 3.8% and false illegals ranging from 1.5% to 1.7%.

VII. Related Work

We are not aware of any other work that focuses specifically on static detection of disassembly errors. The work that is closest to this is that of Kruegel *et al.*, who use statistical analyses of a collection of training executables to choose the likeliest of several alternative disassemblies of an executable [7]. These techniques rely on empirically-determined probability distributions for instructions and instruction pairs, which are augmented with heuristics involving operand characteristics. We experimented using a similar approach but found it difficult to specify a fixed cutoff threshold for opcode and opcode-pair probabilities such that values below the threshold accurately identified incorrect disassemblies and values above it accurately identified correct disassemblies. Furthermore, it is not clear how the heuristics for reasoning about operand characteristics might be systematically derived.

Rieck *et al.* apply machine learning to malware behaviors to develop a classifier for malware [18]. This work differs from ours in that it focuses on high-level features of the runtime behavior of executables while we consider their low-level static characteristics; their goals are also quite different from ours, focusing on determining whether an executable exhibits malware-like behavior, while our work aims specifically at the problem of identifying errors in disassembly. There has also been some research on statistical analysis of byte or opcode distributions in executable files to identify executables that are packed or encrypted [2], [10]. The techniques and goals of these works are very different from ours. In particular, they focus on overall statistical properties of executables taken as a whole, rather than on the more fine-grained focus of our work that examines features of individual instructions.

VIII. Conclusions

Disassembly of executables is an important component of reverse engineering the code to understand its internal working. 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 files.

Acknowledgments

This work was supported in part by the Air Force Office of Scientific Research via grant no. FA9550-07-1-0019, and by the National Science Foundation via grant nos. CNS-0410918 and CNS-0615347.

References

[1] Aspack software. http://www.aspack.com/asprotect.html.

[2] D. Bilar. Opcodes as predictor for malware. *Int. J. Electron. Secur. Digit. Forensic*, 1(2):156–168, 2007.

[3] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions, Malware and Vulnerability Analysis (DIMVA)*, July 2008.

[4] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proc. 12th Usenix Security Symposium*, pages 169–186, August 2003.

[5] A. Danielescu. Anti-debugging and anti-emulation techniques. *CodeBreakers Journal*, 5(1), 2008. http://www.codebreakers-journal.com/.

[6] A. Kapoor. An approach towards disassembly of malicious binaries. Master's thesis, University of Louisiana at Lafayette, 2004.

[7] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proc. 13th USENIX Security Symposium*, August 2004.

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

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

[10] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy*, 5(2):40–45, 2007.

[11] R. J. Mooney. CS 391L: Machine learning: Decision tree learning. `www.cs.utexas.edu/~mooney/cs391L/slides/dtrees.ppt`.

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

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

[14] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *ISPASS '05: Proc. IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pages 10–20, 2005.

[15] I. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *Proc. Usenix Security 2007*, pages 275–290, August 2007.

[16] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[17] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[18] K. Rieck, T. Holz, C. Willems, P. Düsel, and P. Laskow. Learning and classification of malware behavior. In *Proc. Fifth Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2008)*, July 2008.

[19] B. Schwarz, S. K. Debray, and G. R. Andrews. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

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

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