Differentiating Code from Data in x86 Binaries[*]

Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and
Bhavani Thuraisingham

Computer Science Department
University of Texas at Dallas
Richardson, TX 75080
{rhw072000,yan.zhou2,hamlen,muratk,bhavani.thuraisingham}@utdallas.edu

Abstract. Robust, static disassembly is an important part of achieving high coverage for many binary code analyses, such as reverse engineering, malware analysis, reference monitor in-lining, and software fault isolation. However, one of the major difficulties current disassemblers face is differentiating code from data when they are interleaved. This paper presents a machine learning-based disassembly algorithm that segments an x86 binary into subsequences of bytes and then classifies each subsequence as code or data. The algorithm builds a language model from a set of pre-tagged binaries using a statistical data compression technique. It sequentially scans a new binary executable and sets a breaking point at each potential code-to-code and code-to-data/data-to-code transition. The classification of each segment as code or data is based on the minimum cross-entropy. Experimental results are presented to demonstrate the effectiveness of the algorithm.

Keywords: statistical data compression, segmentation, classification, x86 binary disassembly

1 Introduction

Disassemblers transform machine code into human-readable assembly code. For some x86 executables, this can be a daunting task in practice. Unlike Java bytecode and RISC binary formats, which separate code and data into separate sections or use fixed-length instruction encodings, x86 permits interleaving of code and static data within a section and uses variable-length, unaligned instruction encodings. This trades simplicity for brevity and speed, since more common instructions can be assigned shorter encodings by architecture designers. An unfortunate consequence, however, is that hidden instructions can be concealed within x86 binaries by including jump instructions that target the interior of another instruction's encoding, or that target bytes that resemble

[*] This material is based upon work supported by the AFOSR under contracts FA9550-08-1-0044, FA9550-10-1-0088, and FA9550-08-1-0265; by the NIH through grant number 1R01LM009989; and by the NSF through grant numbers Career-0845803, CNS-0964350, and CNS-1016343.

data. This causes these bytes to be interpreted as code at runtime, executing code that does not appear in the disassembly. Malicious code is therefore much easier to conceal in x86 binaries than in other formats. To detect and identify potential attacks or vulnerabilities in software programs, it is important to have a comprehensive disassembly for analyzing and debugging the executable code.

In software development contexts, robust disassembly is generally achieved by appealing to binary debugging information (e.g., symbol/relocation tables) that is generated by most compilers during the compilation process. However, such information is typically withheld from consumers of proprietary software in order to discourage reverse engineering and to protect intellectual property. Thus, debugging information is not available for the vast majority of COTS binaries and other untrusted mobile code to which reverse engineering is typically applied.

Modern disassemblers for x86 binaries therefore employ a variety of heuristic techniques to accurately differentiate bytes that comprise instructions from those that comprise static data. The techniques are heuristic because fully correct x86 disassembly is provably undecidable: Bytes are code if and only if they are reachable at runtime—a decision that reduces to the halting problem.

IDA Pro [9] is widely acknowledged as the best x86 static disassembly tool currently available for distinguishing code from data in arbitrary binaries (cf., [1, 6, 12]). It combines straight-line, heuristic, and execution emulation-based disassembly while also providing an extensive GUI interface and multiple powerful APIs for interacting with the disassembly data. Recent work has applied model-checking and abstract interpretation to improve upon IDA Pro's analysis [12, 13], but application of these technologies is currently limited to relatively small binaries, such as device drivers, for which these aggressive analyses remain tractable. All other widely available disassemblers to our knowledge take a comparatively simplistic approach that relies mainly upon straight-line disassembly, and that therefore requires the user to manually separate code from data during binary analysis. Our tests therefore focus on comparing the accuracy of our algorithm to that of IDA Pro.

Disassembly heuristics employed by IDA Pro include the following:

- *Code entry point.* The starting point for analyzing an executable is the address listed in the header as the code entry point. That address must hold an instruction, and will hopefully lead to successfully analyzing a large portion of the executable.
- *Function prologues and epilogues.* Many function bodies compiled by mainstream compilers begin with a recognizable sequence of instructions that implement one of the standard x86 calling conventions. These byte sequences are assumed by IDA Pro to be the beginnings of reachable code blocks.
- *Direct jumps and calls.* The destination address operand of any static jump instruction that has already been classified as reachable code is also classified as reachable code.
- *Unconditional jumps and returns.* Bytes immediately following a reachable, unconditional jump or return instruction are considered as potential data

bytes. These often contain static data such as jump tables, padding bytes, or strings.

However, despite a decade of development and tuning, IDA Pro nevertheless fails to reliably distinguish code from data even in many non-malicious, non-obfuscated x86 binaries. Some common mistakes include the following:

- *Misclassifying data as returns.* IDA Pro frequently misclassifies isolated data bytes within data blocks as return instructions. Return instructions have a one-byte x86 encoding and are potential targets of computed jumps whose destinations are not statically decidable. This makes them extremely difficult to distinguish from data. IDA Pro therefore often misidentifies data bytes that happen to match the encoding of a return instruction.
- *16-bit legacy instructions.* The x86 instruction set supports legacy 16-bit addressing modes, mainly for reasons of backward compatibility. The vast majority of genuinely reachable instructions in modern binaries are 32- or 64-bit. However, many data bytes or misaligned code bytes can be misinterpreted as 16-bit instructions, leading to flawed disassemblies.
- *Mislabeled padding bytes.* Many compilers generate padding bytes between consecutive blocks of code for alignment purposes. These bytes are not reached by typical runs, nor accessed as data, so their proper classification is ambiguous. IDA Pro typically classifies them as data, but this can complicate some code analyses by introducing many spurious code-data boundaries in the disassembly. In addition, these bytes can later become reachable if the binary undergoes hotpatching [10]. We therefore argue that these bytes are more properly classified as code.
- *Flows from code to data.* IDA Pro disassemblies frequently contain data bytes immediately preceded by non-branching or conditionally branching instructions. This is almost always an error; either the code is not actually reachable (and is therefore data misidentified as code) or the data is reachable (and is therefore code misidentified as data). The only exception to this that we have observed in practice is when a call instruction targets a non-returning procedure, such as an exception handler or the system's process-abort function. Such call instructions can be immediately followed by data.

To provide a rough estimate of the classification accuracy of IDA Pro, we wrote scripts in IDAPython [7] that detect obvious errors made by IDA Pro in its disassemblies. Table 1 gives a list of the executables we tested and counts of the errors we identified for IDA Pro 5.5. The main heuristic we used to identify errors is the existence of a control-flow from code to data. Certain other errors were identified via manual inspection. It is interesting to note that most programs compiled using the GNU family of compilers have little to no errors in their IDA Pro disassemblies. This is probably because GNU compilers tend to yield binaries in which code and data are less interleaved, and they perform fewer aggressive binary-level optimizations that can result in code that is difficult to disassemble.

In this paper, we present a disassembly algorithm that combines the heuristics manually applied by experts during reverse engineering and a language model

Table 1. Statistics of IDA Pro 5.5 disassembly errors

File Name	Instructions	Mistakes
Mfc42.dll	355906	1216
Mplayerc.exe	830407	474
RevelationClient.exe	66447	36
Vmware.exe	364421	183

that can capture both short-range and long-range correlations between byte sequences. Experimental results demonstrate that our algorithm can identify and successfully label a large number of code sequences that are missed by IDA Pro.

2 A Language Model for Disassembling x86 Executables

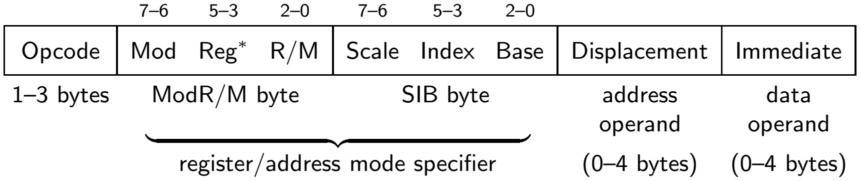
Without any debugging information at our disposal, we treat any given x86 executable as a string of arbitrary unsigned bytes. Our first task is to segment the single string into consecutive subsequences that are either code or data. A code-to-code, code-to-data, or data-to-code transition event occurs at each boundary between different instructions or between code and data.

The Intel architecture manual [11] specifies the decoding of each x86 instruction if the starting point for the instruction is known. Unfortunately, when code and data are interleaved it is not obvious whether a byte is the start of an instruction, the interior of an instruction, or a non-instruction (i.e., data). To tackle this problem we first decide whether a sequence of bytes is more likely to be code or data. The executable is then segmented using the opcodes defined in the Intel instruction encoding specification. Since we are unable to ensure a perfect segmentation, our next task is to classify each subsequence as code or data. Both tasks involve a context-based language model. We next formally describe each task and discuss the language model used in our disassembly algorithm.

2.1 Code Segmentation

In this section we first briefly review the x86 machine instruction set. We then define the code segmentation problem and present our algorithm to solve the problem.

Instruction Encodings Figure 1 shows the x86 machine instruction binary format [11]. Instructions begin with 1–3 *opcode* bytes that identify the instruction. Instructions with operands are then followed by an *addressing form specifier* (ModR/M) byte that identifies register or memory operands for the instruction. Some addressing forms require a second *scale-index-base* (SIB) byte that specifies a memory addressing mode. The addressing mode essentially encodes a short formula that dynamically computes the memory operand at runtime. For example, addressing mode `[eax*4]+disp32` references a memory address obtained by multiplying the contents of the `eax` register by 4 and then adding a 32-bit *displacement* constant. The displacement, if present, comes after the SIB byte. Finally, immediate operands (constants) are encoded last and have a width of up to 4 bytes (on 32-bit architectures).



*The Reg field is sometimes used as an opcode extension field.

Fig. 1. The x86 machine instruction format.

In addition to this complicated instruction format, there are a number of prefix bytes that may precede the opcode bytes, all eleven of which may be used in combination. Some of these prefix bytes, if present, affect the length of the succeeding instruction's encoding by temporarily changing the default operand widths.

A few x86 machine instructions have multiple different correct representations at the assembly level. Most notable is the floating point `WAIT` instruction, which can either be interpreted as an opcode prefix for the instruction it precedes, or as a separate instruction in its own right. We adopt the former interpretation in our treatment, since it makes for a more compact assembly representation.

Problem Definition We define the *tagging problem* as follows: Given a non-empty input string X over an alphabet Σ, find a set of transition events $\mathcal{T}^* = \{\$_1, \ldots, \$_M\}$ such that $\mathcal{T}^* = \arg\max_{\mathcal{T}} f(X, \mathcal{T})$, where $\$_i$ at position $i < |X|$ marks a transition event e in X, $\mathcal{T}$ denotes any possible set of transition events, and f is a function that measures the likelihood that X is tagged correctly.

The tagging problem resembles the word segmentation problem in some natural languages where no clear separations exist between different words [15]. In the word segmentation problem, the task is to find correct separations between sequences of characters to form words. In the tagging problem, our objective is to find separations between different instructions, and often between instructions and data as well. In both problems, resolving ambiguities is the major challenge. For example, a byte sequence `E8 F9 33 6A 00` can be a 5-byte call instruction (opcode `E8`), or three bytes of data followed by a push instruction (opcode `6A`). Ambiguities can only be resolved through investigating their surrounding context.

Solutions to the tagging problem must also successfully identify and ignore "noise" in the form of padding bytes. Padding bytes are neither executed as code nor accessed as data on any run of the executable, so their classification is ambiguous. However, reliably distinguishing these padding sequences from true code and data is highly non-trivial because the same sequence of bytes often appears as both code and padding within the same executable. For example, the instruction

```
8D A4 24 00 00 00 00    lea esp, [esp+0x0]
```

is semantically a *no-operation* (NOP), and is therefore used as padding within some instruction streams to align subsequent bytes to a cache line boundary, but is used in other instruction streams as a genuinely reachable instruction. Another common use of semantic NOPs is to introduce obfuscation to hide what the program is doing.

In general, code and data bytes may differ only in their locations in the sequence, not in their values. Any byte sequence that is code could appear as data in an executable, even though it should statistically appear much more often as code than data. Not every data sequence can be code, however, since not all byte sequences are legitimate instruction encodings.

The Tagging Algorithm There are two components in our tagging algorithm: an instruction reference array and a utility function. The reference array stores the length of an instruction given the bytes of an opcode (and the existence of length-relevant prefix bytes). The utility function estimates the probability that a byte sequence is code. We estimate the probability using a context-based language model built from pre-tagged x86 executables.

Instruction Reference Array. From the x86 instruction decoding specification we derive a mapping from the bytes of an opcode to the length of the instruction. This is helpful in two respects: First, it marks a definite ending of an instruction that allows us to move directly to the next instruction or data. Second, it tells us when a series of bytes is undefined in the x86 instruction set, which means that the current byte cannot be the beginning of an instruction. We tested our code against more than ten million instructions in the IDA Pro disassembler and had 100% accurate instruction lengths.

Utility Function. The utility function helps predict whether a byte sequence is code or data in the current context. If the current byte sequence is unlikely to be code, our tagging algorithm moves to the next byte sequence. If we predict that the byte sequence is code, we look up the length of the instruction in the instruction reference array and move to the next byte sequence. The following two properties express the desired relationship between the utility function and its input byte sequence.

Property 1. A byte sequence bordered by transitions is tagged as code (resp., data) if its utility as code (resp., data) is greater than its utility as data (resp., code).

Property 2. A transition between two byte sequences S_A and S_B entails a semantic ordering in machine code: $f(S_B|S_A) \geq f(S_B|S_*)$, where S_* is any subsequence but S_A in a given binary, and f is the utility function.

Our utility function estimates the likelihood of a transition event using context-based analysis. We collect context statistics from a set of pre-tagged binaries in the training set. In a pre-tagged binary, code-code and code-data/data-code transitions are given. Two important forms of information are yielded by pre-tagged binaries. First, they provide semantic groupings of byte sequences that are either code or data; and second, they provide a semantic ordering between two subsequences, which predicts how likely a subsequence is followed by another. To correctly tag an input hex string, both pieces of information are important. This calls for a language model that

- can capture local coherence in a byte sequence, and
- can capture long-range correlations between two adjacent subsequences—i.e., subsequences separated by a code-code or code-data/data-code transition.

Several modern statistical data compression models [14] are known for their context-based analysis. These data models can work directly on any raw input regardless of source and type. We use the current state of the art data compression model as our language model. Before we discuss the details of the language model, we give the tagging algorithm in Algorithm 1.

Algorithm 1: Tagging

```
Input: $x_0 \ldots x_i \ldots x_{n-1}$                        // input string of bytes
       $M_c$                                                   // language model
Output: $x_0 \ldots x_i | x_{i+1} \ldots x_j | \cdots | x_k \ldots x_{n-1}$   // segmented string
$t \leftarrow 0$
while $t < n$ do
    $\ell \leftarrow 0$
    if $x_t \in M_c$ then
        $\ell \leftarrow codeLength(x_t \ldots x_{\min\{t+4,n-1\}})$   // lookup instruction length
    if $(\ell = 0) \vee (t + \ell > n)$ then $\ell \leftarrow 1$      // tag as possible data
    print $x_t \ldots x_{t+\ell-1}$                                 // output the segment
    $t \leftarrow t + \ell$
```

2.2 Context-based Data Compression Model

The compression model we use to store context statistics is *predication by partial matching* (PPM) [4, 5, 3]. The theoretical foundation of the PPM algorithm is the kth order Markov model, where k constrains the maximum order context based on which a symbol probability is predicted. PPM models both short-range and long-range correlations among subsequences by using dynamic context match. The context of the ith symbol x_i in an input string is the previous $i-1$ symbols. Its kth order context c_i^k includes only the k prior symbols. To predict the probability of seeing x_i in the current location of the input, the PPM algorithm first searches for a match of c_i^k in the context tree. If a match is found, $p(x_i|c_i^k)$ is returned as the symbol probability. If such a match does not exist in the context tree, an *escape event* is recorded and the model falls back to a lower-order context c_i^{k-1}. If a match is found, the following symbol probability is returned:

$$p(x_i|c_i^k) = p(Esc|c_i^k) \cdot p(x_i|c_i^{k-1})$$

where $p(Esc|c_i^k)$ is the escape probability conditioned on context c_i^k. The *escape probability* models the probability that x_i will be found in the lower-order context. This process is repeated whenever a match is not found until an order-0 context has been reached. If x_i appears in the input string for the first time, a uniform probability of distinct symbols that have been observed so far will be returned. Therefore, the probability of x_i in a string of input is modeled as follows:

$$p(x_i|c_i^k) = \begin{cases} \left(\prod_{j=k'+1}^{k} p(Esc|c_i^j)\right) \cdot p(x_i|c_i^{k'}) & \text{if } k \geq 0 \\ \frac{1}{|A|} & \text{if } k = -1 \end{cases}$$

where $k' \leq k$ is the context order when the first match is found for x_i, and $|A|$ is the number of distinct symbols seen so far in the input. If the symbol is not predicted by the order-0 model, a probability defined for the order -1 context is predicted.

The PPM model predicts symbol probabilities. To estimate the probability of a sequence of symbols, we compute the product of the symbol probabilities in the sequence. Thus, given a data sequence $X = x_1 x_2 \ldots x_d$ of length d, where x_i is a symbol in the alphabet, the probability of seeing the entire sequence given a compression model M can be estimated as

$$p(X|M) = \prod_{i=1}^{d} p(x_i|x_{i-k}^{i-1})$$

where $x_i^j = x_i x_{i+1} x_{i+2} \ldots x_j$ for $i < j$.

We use the above probability estimate as our utility function. We build two compression models M_c and M_d from the pre-tagged binaries in the training set: M_c is built from tagged instructions and M_d is built from tagged data. Given a new binary executable e and a subsequence e_i in e,

$$M_c = \{e_i \,|\, p(e_i|M_c) > p(e_i|M_d)\}$$

2.3 Classification

After tagging the transitions in the executable, we have segments of bytes. Even though the tagging algorithm outputs each segment either as code or data, we cannot assume this preliminary classification is correct because some data bytes may match legitimate opcodes for which a valid instruction length exists in the reference array. The tagging algorithm will output this segment as code even though it is data. Therefore, we need to reclassify each segment as data or code.

Our classification algorithm makes use of the aforementioned language model and several well known semantic heuristics. The language models are also used in the tagging algorithm. The heuristics are adapted from those used by human experts for debugging disassembly errors. We first discuss the language model-based classification module followed by the semantic heuristics.

Classification Using Language Model Classifying byte sequences is a binary classification problem. We reuse the two compression models built for tagging. Recall that model M_c is built from pre-tagged code and model M_d is built from the pre-tagged data in the training set. To classify a byte sequence B, we compute a log likelihood of B using each data model $\alpha \in \{c, d\}$:

$$p(B|M_\alpha) = -\log \prod_{i=1}^{|B|} p(b_i|b_{i-k}^{i-1}, M_\alpha)$$

where M_α is the compression model associated with class α, $|B|$ is the length of byte sequence B, sequence $b_{i-k}, \ldots, b_i$ is a subsequence in B, and k is the length of the context. The class membership α of B is predicted by minimizing the cross entropy [16, 2]:

$$\alpha = \operatorname*{arg\,min}_{\alpha \in \{c,d\}} -\frac{1}{|B|} p(B|M_\alpha)$$

Classification Using Heuristics In addition to our context-based language models, certain semantic heuristics are helpful in determining an accurate class membership of an x86 byte sequence. Reverse engineers rely heavily upon such heuristics when manually correcting flawed disassemblies.

Word data tables. Many static data blocks in code sections store tables of 4-byte integers. Often the majority of 4-byte integers in these tables have similar values, such as when the table is a method dispatch or jump table consisting of code addresses that mostly lie within a limited virtual address range. One way to quickly identify such tables is to examine the distribution of byte values at addresses that are 1 less than a multiple of 4. When these high-order bytes have low variance, the section is likely to be a data table rather than code, and is classified accordingly.

16-bit addressing modes. When classifying a byte sequence as code yields a disassembly densely populated by instructions with 16-bit operands (and the binary is a 32-bit executable), this indicates that the sequence may actually be data misclassified as code. Modern x86 architectures support the full 16-bit instruction set of earlier processor generations for backward compatability reasons, but these legacy instructions appear only occasionally in most modern 32-bit applications. The 16-bit instructions often have short binary encodings, causing them to appear with higher frequency in randomly generated byte sequences than they do in actual code.

Data after unconditional jumps. Control-flows from code to data are almost always disassembly errors; either the data is reachable and is therefore code, or the code is actually unreachable and is therefore data. Thus, data inside of a code section can only occur at the very beginning of the section or after a branch instruction—usually an unconditional jump or return instruction. It can occasionally also appear after a call instruction if the call never returns (e.g., the call targets an exception handler or process-abort function). This observation gives rise to the following heuristics:

- If an instruction is a non-jump, non-return surrounded by data, it is reclassified as data.
- If a byte sequence classified as data encodes an instruction known to be a semantic NOP, it is reclassified as code.

3 Experimental Results

We tested our disassembly algorithm on the 11 real-world programs listed in Table 2. In each experiment, we used 10 of the programs to build the language models and the remaining one for testing. All the executables are pre-tagged using IDA Pro; however, IDA Pro yields imperfect disassemblies for all 11 executables. Some instructions it consistently labels as data, while others—particularly those that are semantic NOPs—it labels as data or code depending on the context. This leads to a noisy training set.

Since we lack perfect disassemblies of any of these programs, evaluation of the classification accuracy of each algorithm is necessarily based on a manual comparison of the disassembly results. When the number of classification disagreements is large, this can quickly exceed the human processing limit. However, disagreements in which one algorithm identifies a large, contiguous code section missed by the other are relatively easy to verify by manual inspection. These constituted the majority of the disagreements, keeping the evaluation tractable.

3.1 Tagging Results

We first report the accuracy of our tagging algorithm. Inaccuracies can take the form of code misclassified as data (false negatives) and data misclassified as code

Table 2. Software programs for testing

File Name	File Size (K)	Code (K)	Data (K)	Transitions
`7zFM.exe`	379	271	3.3	1379
`notepad.exe`	68	23	8.6	182
`DosBox.exe`	3640	2947	67.2	15355
`WinRAR.exe`	1059	718	31.6	5171
`Mulberry.exe`	9276	4632	148.2	36435
`scummvm.exe`	11823	9798	49.2	47757
`emule.exe`	5624	3145	119.5	24297
`Mfc42.dll`	1110	751	265.5	15706
`Mplayerc.exe`	5858	4044	126.1	28760
`RevelationClient.exe`	382	252	18.4	1493
`Vmware.exe`	2675	1158	87.3	18259

(false positives). Both can have potentially severe consequences in the context of reverse engineering for malware defense. False negatives withhold potentially malicious code sequences from expert analysis, allowing attacks to succeed; false positives increase the volume of code that experts must examine, exacerbating the difficulty of separating potentially dangerous code from benign code. We therefore compute the tagging accuracy as

$$accuracy = 1 - \frac{false\ negatives + false\ positives}{total\ number\ of\ instructions}$$

where false positives count the number of instructions erroneously disassembled from data bytes.

As can be seen in Table 3 we were able to tag 6 of the 11 binaries with 100% accuracy. For the remaining 5, the tagging errors were mainly caused by misclassification of small word data tables (see §2.3) consisting of 12 or fewer bytes. Our heuristic for detecting such tables avoids matching such small tables in order to avoid misclassifying short semantic NOP sequences that frequently pad instruction sequences. Such padding often consists of 3 identical 4-byte instructions, which collectively resemble a very short word data table.

3.2 Classification Results

To evaluate the classification accuracy we took the output of our tagging algorithm and ran each segment through the language model to get its class membership. Table 4 shows the classification results of our disassembly algorithm. False positives (FP), false negatives (FN), and overall classification accuracy is listed for each disassembler. False positives are subsequences that are data misclassified as code and false negatives are those that are code misclassified as data. As can be seen in Table 4 we were able to classify five of the 11 binaries with 100% accuracy.

Table 3. Tagging accuracy

File Name	Errors	Total	Tagging Accuracy
7zFM.exe	0	88164	100%
notepad.exe	0	6984	100%
DosBox.exe	0	768768	100%
WinRAR.exe	39	215832	99.982%
Mulberry.exe	0	1437950	100%
scummvm.exe	0	2669967	100%
emule.exe	117	993159	99.988%
Mfc42.dll	0	355906	100%
Mplayerc.exe	307	830407	99.963%
RevelationClient.exe	71	66447	99.893%
Vmware.exe	16	364421	99.998%

Table 4. A comparison of mistakes made by IDA Pro and by our disassembler

	IDA Pro 5.5			Ours		
File Name	FP	FN	Accuracy	FP	FN	Accuracy
7zFM.exe	0	1	99.999%	0	0	100%
notepad.exe	4	0	99.943%	0	0	100%
DosBox.exe	0	26	99.997%	0	0	100%
WinRAR.exe	0	23	99.989%	0	39	99.982%
Mulberry.exe	0	202	99.986%	0	0	100%
scummvm.exe	0	65	99.998%	0	0	100%
emule.exe	0	681	99.931%	0	117	99.988%
Mfc42.dll	0	1216	99.658%	0	47	99.987%
Mplayerc.exe	0	2065	99.751%	0	307	99.963%
RevelationClient.exe	0	1781	97.320%	0	71	99.893%
Vmware.exe	0	183	99.950%	0	45	99.988%

3.3 eMule Case Study

To show some of the specific differences between decisions made by IDA Pro's disassembler and our approach, we here present a detailed case study of eMule, a popular peer-to-peer file sharing program. Case studies for other executables in our test suite are similar to that presented here. Table 5 illustrates examples in which IDA Pro classified bytes were code but our disassembler determined that they were data, or vice versa. In the table, *db* is an assembly directive commonly used to mark data bytes in a code listing. To identify all discrepancies, we stored all instructions from both disassemblies to text files with code/data distinguishers before every instruction. We then used `sdiff` to find the differences. The cases in Table 5 summarize all of the different kinds of discrepancies we discovered.

IDA Pro makes heavy use of heuristic control-flow analysis to infer instruction start points in a sea of unclassified bytes. Thus, its classification of bytes imme-

Table 5. Disassembly discrepancies between IDA Pro and our disassembler for eMule

		Example Disassemblies	
Case	**Description**	**IDA Pro 5.5**	**Ours**
1	padding after a non-returning call	`call ExceptionHandler`	`call ExceptionHandler`
		db (1–9 bytes)	*code (1–9 bytes)*
		`function_start`	`function_start`
2	calls misidentified as non-returning	`call GetDLGItem`	`call GetDLGItem`
		db 88h 50h	`mov edx, [eax+1Ch]`
		`sbb al, 8Bh`	
3	repetitive instruction sequences	*db (4 bytes)*	`push 0`
			`push 0`
		`push 0`	`push 0`
		`push 0`	`push 0`
		`call 429dd0h`	`call 429dd0h`
4	missed computed jump targets	*db (12 bytes)*	`mov eax, large fs:0`
			`mov edx, [esp+8]`
			`push FFFFFFFFh`
		`push offset 41CC30h`	`push offset 41CC30h`
5	false computed jump targets	`push ecx`	`push ecx`
		db FFh	`call 7DFAB4h`
		`adc eax, 7DFAB4h`	
		`mov ebp, eax`	`mov ebp, eax`
		db 8Bh	`mov eax, [ebx+0DCh]`
		`sbb esp, 0`	`mov ecx, [eax+4]`
		db (13 bytes)	`cmp ecx, esi`
			`jle loc_524D61`
		`test esi, esi`	`test esi, esi`
6	missed opcode prefixes	`push offset 701268h`	`push offset 701268h`
		db 64h	`mov eax, large fs:0`
		`mov eax, large ds:0`	
7	code following unconditional branches	`jmp 526396h`	`jmp 526396h`
		db 8Bh	`mov ecx, 9CAF08h`
		`or eax, 9CAF08h`	
8	code following returns	`retn`	`retn`
		db C4h 83h	`add esp, 2Ch`
		`sub al, CDh`	`int 6`
		`push es`	
9	code following conditional branches	`jz 52518Fh`	`jz 52518F`
		db 8Bh	`mov ecx, 9CAF04h`
		`or eax, 9CAF04h`	

diately following a call instruction depends on its estimate of whether the called method could return. For example, Case 1 of Table 5 shows a non-returning call to an exception handler. The call is immediately followed by padding bytes that serve to align the body of the next function. These bytes are also legitimate (but unreachable) instructions, so could be classified as data or code (though we argue in §1 that a code classification is preferable). However, this control-flow analysis strategy leads to a classification error in Case 2 of the table, wherein IDA Pro incorrectly identifies method `GetDLGItem` as non-returning and therefore fails to disassemble the bytes that follow the call. Our disassembler correctly identifies both byte sequences as code. Such scenarios account for about 20 of IDA Pro's disassembly errors for eMule.

Case 3 of Table 5 illustrates a repetitive instruction sequence that is difficult to distinguish from a table of static data. IDA Pro therefore misidentifies some of the bytes in this sequence as data, whereas our algorithm correctly identifies all as code based on the surrounding context.

Many instruction sequences in x86 binaries are only reachable at runtime via dynamically computed jumps. These sequences are difficult to identify by control-flow analysis alone since the destinations of dynamic jumps cannot be statically predicted in general. Case 4 is an example where IDA Pro fails to identify a computed jump target and therefore fails to classify the bytes at that address as code; however, our disassembler finds and correctly disassembles the instructions.

Misidentifying non-jump targets as possible targets leads to a different form of disassembly error. Case 5 illustrates an example in which an early phase of IDA Pro's analysis incorrectly identifies the interior byte of an instruction as a possible computed jump destination (probably because some bytes in a data section happened to encode that address). The bytes at that address disassemble to an `adc` instruction that turns out to be misaligned with respect to the surrounding sequence. This leads to an inconsistent mix of code and data that IDA Pro cannot reconcile because it cannot determine which interpretation of the bytes is correct. In contrast, our algorithm infers the correct instruction sequence, given in the rightmost column of the table.

Some instructions include prefix bytes, as discussed in §2.1. The suffix without the prefix bytes is itself a valid instruction encoding. IDA Pro's analysis sometimes misses these prefix bytes because it discovers the suffix encoding first and treats it as a self-contained instruction. This leads to the disassembly error depicted in Case 6 of the table. Our approach avoids this kind of error in all cases.

Cases 7–8 of the table illustrate disassembly errors in which IDA Pro fails to identify code bytes immediately following unconditional jumps and returns. These too are a consequence of relying too heavily on control-flow analysis to discover code bytes. Occasionally these errors even appear after conditional jumps, as shown in Case 9. It is unclear why IDA Pro makes this final kind of mistake, though we speculate that it may be the result of a dataflow analysis that incorrectly infers that certain conditional branches are always taken and therefore never fall through. Use of conditional branches as unconditional jumps is a common malware obfuscation technique that this analysis may be intended to counter. However, in this case it backfires and leads to an incorrect disassembly. Our method yields the correct disassembly on the right.

4 Conclusion

We developed and evaluated an automated disassembler using context-aware language models to separate instructions from instructions and code from data. Each segment in the resulting byte sequence is then separately classified as code or data. Evaluation of the technique demonstrates that our algorithm consistently yields more accurate disassemblies than the IDA Pro disassembler, which is widely regarded as the best commercial disassembly tool currently available.

Future work includes blending more sophisticated heuristics into our learning model, and trying block entropy approaches to better estimate the boundary between code and data.

In addition, larger-scale evaluation of our results could be facilitated by automating more of the evaluation process. One possible approach is to generate test binaries with perfect labels by using compiler options that artificially separate code from data, or that yield binary debugging information that can be used to infer correct labels. Unfortunately, most compilers and compiler modes that yield binaries for which the disassembly task is non-trivial are specifically those compilers that are not easy to modify (e.g., non-open source compilers) and those modes that do not support debugging (e.g., highly optimizing release modes). Pursuing this approach therefore requires identifying a suitable compiler.

We also plan to apply our disassembly technique to support more effective and reliable analysis and instrumentation of x86 binaries without source code for security purposes [8].

References

1. Balakrishnan, G., Gruian, R., Reps, T., Teitelbaum, T.: CodeSurfer/x86—a platform for analyzing x86 executables. In: Proceedings of the 14th International Conference on Compiler Construction (CC). pp. 250–254 (2005)
2. Bratko, A., Cormack, G.V., Filipič, B., Lynam, T.R., Zupan, B.: Spam filtering using statistical data compression models. Journal of Machine Learning Research 7, 2673–2698 (2006)
3. Cleary, J.G., Teahan, W.J.: Unbounded length contexts for PPM. The Computer Journal 40(2/3), 67–75 (1997)
4. Cleary, J.G., Witten, I.H.: Data compression using adaptive coding and partial string matching. IEEE Transactions on Communications 32(4), 396–402 (1984)
5. Cormack, G.V., Horspool, R.N.: Data compression using dynamic Markov modeling. The Computer Journal 30(6), 541–550 (1987)
6. Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. No Starch Press, Inc., San Francisco, California (2008)

7. Erdélyi, G.: IDAPython: User scripting for a complex application. Bachelor's thesis, EVTEK University of Applied Sciences (2008)
8. Hamlen, K.W., Mohan, V., Wartell, R.: Reining in Windows API abuses with inlined reference monitors. Tech. Rep. UTDCS-18-10, The University of Texas at Dallas, Richardson, Texas (June 2010)
9. Hex-Rays: The IDA Pro disassembler and debugger. `www.hex-rays.com/idapro` (2011)
10. Hunt, G., Brubacher, D.: Detours: Binary interception of Win32 functions. In: Proceedings of the 3rd USENIX Windows NT Symposium (WINSYM). pp. 14–21 (1999)
11. Intel: Intel® 64 and IA-32 Architectures Software Developer's Manual, vol. 2A & 2B: Instruction Set Reference. Intel Corporation (2011)
12. Kinder, J., Veith, H.: Jakstab: A static analysis platform for binaries. In: Proceedings of the 20th International Conference on Computer Aided Verification (CAV). pp. 423–427 (2008)
13. Kinder, J., Zuleger, F., Veith, H.: An abstract interpretation-based framework for control flow reconstruction from binaries. In: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). pp. 214–228 (2009)
14. Moffat, A., Turpin, A.: Compression and Coding Algorithms. Kluwer Academic Publishers, Boston/London (2002)
15. Teahan, W.J., Wen, Y., McNab, R.J., Witten, I.H.: A compression-based algorithm for Chinese word segmentation. Computational Linguistics 26(3), 375–393 (2000)
16. Teahan, W.J.: Text classification and segmentation using minimum cross-entropy. In: Proceedings of the 6th International Conference on Computer-Assisted Information Retrieval (RIAO). pp. 943–961 (2000)