Shingled Graph Disassembly: Finding the Undecideable Path

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

Computer Science Department
The University of Texas at Dallas
{rhw072000,yan.zhou2,hamlen,muratk}@utdallas.edu

Technical Report UTDCS-12-13

June 10, 2013

Abstract

A probabilistic finite state machine approach to statically disassembling x86 machine language programs is presented and evaluated. Static disassembly is a crucial prerequisite for software reverse engineering, and has many applications in computer security and binary analysis. The general problem is provably undecidable because of the heavy use of unaligned instruction encodings and dynamically computed control flows in the x86 architecture. Limited work in machine learning and data mining has been undertaken on this subject. This paper shows that semantic meanings of opcode sequences can be leveraged to infer similarities between groups of opcode and operand sequences. This empowers a probabilistic finite state machine to learn statistically significant opcode and operand sequences in a training corpus of disassemblies. The similarities demonstrate the statistical significance of opcodes and operands in a surrounding context, facilitating more accurate disassembly of new binaries. Empirical results demonstrate that the algorithm is more efficient and effective than comparable approaches used by state-of-the-art disassembly tools.

1 Introduction

Statistical data mining techniques have found wide application in domains where statistical information is valuable for solving problems. Examples include computer vision, web search, natural language processing, and more. A recent addition to this list is *static disassembly* [8, 13]. Disassembly is the process of translating byte sequences to human-readable assembly code. Such translation is often deemed a crucial first step in software reverse engineering and analysis.

Although all binary-level debuggers perform *dynamic disassembly* to display assembly code for individual runs of target programs, the much more challenging task of static disassembly attempts to provide assembly code for all possible runs (i.e., all reachable instructions). Static disassembly is therefore critical for analyzing code with non-trivial control-flows, such as branches and loops. Example applications include binary code optimization, reverse engineering legacy code, semantics-based security analysis, malware analysis, intrusion detection, and digital forensics. Incorrectly disassembled binaries often lead to incorrect analyses, and therefore bugs or security vulnerabilities in mission-critical systems.

Static disassembly of binaries that target Intel-based architectures is particularly challenging because of the architecture's heavy use of variable-length, unaligned instruction encodings, dynamically computed control-flows, and interleaved code and data. *Unalignment* refers to the fact that Intel chipsets consider all memory addresses to be legal instruction starting points. When some programs compute the destinations of jumps dynamically using runtime pointer arithmetic, statically deciding which bytes are part of reachable instructions and which are (non-executed) static data reduces from the halting problem. As a result, the static disassembly problem for Intel architectures is provably Turing-undecidable in general.

Production-level disassemblers and reverse engineering tools have therefore applied a long history of evolving heuristics to generate best-guess disassemblies. Such heuristics include fall-through disassembly, various control-flow and dataflow analyses, and compiler-specific pattern matching. Unfortunately, even after decades of tuning, these heuristics often fail even for non-obfuscated, non-malicious, compiler-generated software. As a result, human analysts are often forced to laboriously guide the disassembly process by hand using an interactive disassembler [1]. When binaries are tens or hundreds of megabytes in size, the task quickly becomes intractable.

Our recent past work is the first to apply machine learning and data mining to address this problem [13]. The approach uses statistical data compression techniques to reveal the semantics of a binary in its assembly form, yielding a segmentation of code bytes into assembly instructions and a differentiation of data bytes from code bytes. Although the technique is effective and exhibits improved accuracy over the best commercial disassembler currently available [2], the compression algorithm suffers high memory usage. Thus, training on large corpora can be very slow compared to other disassemblers.

In this paper, we present an improved disassembly technique that is both more effective and more efficient. Rather than relying on high-order context semantic information (which leads to long training times), we leverage a finite state machine with transitional probabilities to infer likely execution paths through a sea of bytes.

Our main contributions include a graph-based static disassembly technique; a simple, efficient, but effective disassembler implementation; and an empirical demonstration of the effectiveness of the approach.

Our high-level strategy involves two linear passes: a preprocessing step which recovers a conservative superset of potential disassemblies, followed by a filtering step in which a state machine selects the best disassembly from the possible candidates. While the resulting disassembly is not guaranteed to be fully correct (due to the undecidability of the general problem), it is guaranteed to avoid certain common errors that plague mainstream disassemblers. Our empirical analysis shows our simple, linear approach is faster and more accurate than the observably quadratic-time approaches adopted by other disassemblers.

The rest of the paper proceeds as follows. Section 2 discusses related work in static disassembly. Section 3 presents our graph-based static disassembly technique. Section 4 presents experimental results, and Section 5 concludes and suggests future work.

2 Related Work

Existing disassemblers mainly fall into three categories: linear sweep disassemblers, recursive traversal disassemblers, and the hybrid approach. The GNU utility *objdump* [9] is a popular example of the linear sweep approach. It starts at the beginning of the text segment of the binary to be disassembled, decoding one instruction at a time until everything in executable sections is decoded. This type of disassembler is prone to errors when code and data bytes are interleaved within some segments. Such interleaving is typical of almost all production-level Windows binaries generated by non-GNU compilers.

IDA Pro [1, 2] follows the recursive traversal approach. Unlike linear sweep disassemblers, it decodes instructions by traversing the static control flow of the program, thereby skipping data bytes that may punctuate the code bytes. However, not all control flows can be predicted statically. When the control flow is constructed incorrectly, some reachable code bytes are missed, resulting in disassemblies that omit significant blocks of code.

The hybrid approach [10] combines linear sweep and recursive traversal to detect and locate disassembly errors. The basic idea is to disassemble using the linear sweep algorithm and verify the output using the recursive traversal algorithm. While this helps to eliminate some disassembly errors, in general it remains prone to the shortcomings of both techniques. That is, when the sweep and traversal phases disagree, there is no clear indication of which is correct; the ambiguous bytes therefore receive an error-prone classification.

The Jakstab fully configurable binary analysis platform [5, 7] is a recent effort to overcome these historic shortcomings by statically resolving computed jump destinations to construct accurate control-flow graphs. The platform implements multiple rounds of disassembly interleaved with dataflow analysis. In each round, the output assembly instructions are translated to an intermediate representation, from which the platform builds a more accurate control-flow graph for dataflow analysis. The results from the dataflow analysis are then used to resolve computed jump targets. This iterative approach exhibits superior accuracy over commercial tools like IDA Pro [2]. However, Jakstab depends on a fixed-point iteration algorithm that is worst-case exponential in the size of the binary being analyzed. It therefore has only been successfully applied to relatively small binaries (e.g., drivers) and does not scale well to full-sized COTS binaries, which are often ten or one hundred times larger. For example, Jakstab requires almost 40 minutes to disassemble a 100K Windows floppy driver [6].

Our recent machine learning- and data mining-based approach to the disassembly problem [13] avoids error-prone control-flow analysis heuristics in favor of a three-phase approach: First, executables are segmented into subsequences of bytes that constitute valid instruction encodings as defined by the architecture [3]. Next, a language model is built from the training corpus with a statistical data model used in modern data compression. The language model is used to classify the segmented subsequence as code or data. Finally, a set of pre-defined heuristics refines the classification results. The experimental results demonstrate substantial improvements over IDA Pro's traversal-based approach. However, it has the disadvantage of high memory usage due to the large statistical compression model. This significantly slows the disassembly process relative to simple sweep and traversal disassemblers.

Machine-learning has also been applied to statically identify errors in disassembly listings [8]. Incorrect disassemblies are typically statistically different from correct disassemblies. Based on this observation, a decision tree classifier can be trained using a set of correct and incorrect disassemblies. The classifier is then used to detect errors such as extraneous opcodes and operands, as well as nonexistent branch target addresses. The experimental results demonstrate that the decision tree classifiers can correctly identify the majority of the disassembly errors in test files while returning relatively few false positives.

Our disassembly algorithm presented in this paper instead adopts a *probabilistic finite state machine* (FSM) [11, 12] approach. FSMs are widely used in areas such as computational linguistics, speech processing, and gene sequencing. Although the transitions of probabilistic FSMs are non-deterministic, they are labeled with probabilities given training data. For any given string, there is more than one trace through the FSM. By querying the FSM, the likelihood of each trace can be computed. Our approach builds FSMs from training corpora. For each new binary

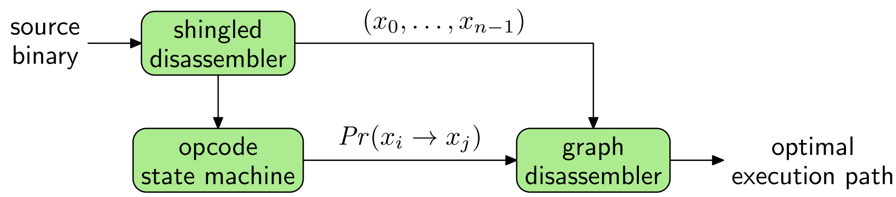

Figure 1: Disassembler architecture

executable, FSM traces reveal probable paths of reachable opcode and operand sequences in an unlabeled byte stream.

3 Disassembler Design

Our machine learning approach to disassembly frames the disassembly problem as follows:

Problem Definition Given an arbitrary string of bytes, which subset of the bytes is the most probable set of potentially reachable instruction starting points, where "probable" is defined in terms of a given corpus of correct binary disassemblies?

Figure 1 shows the architecture of our disassembly technique. It consists of a *shingled disassembler* that recovers the (overlapping) building blocks (*shingles*) of all possible valid execution paths, a finite state machine trained on binary executables, and a graph disassembler that traces and prunes the shingles to output the maximum-likelihood classification of bytes as instruction starting points, instruction non-starting points, and data.

3.1 Shingled Disassembler

Since computed branch instructions in x86 have their targets established at runtime, every byte within the code section can be a target and thus must be considered as executable code. This aspect of the x86 architecture allows for *instruction aliasing*, the ability for two instructions to overlap each other. Therefore, we refer to a disassembler that retains all possible execution paths through a binary as a shingled disassembler.

Definition 1. ***Shingle***
A shingle *is a consecutive sequence of bytes that decodes to a single machine instruction. Shingles may overlap.*

The core functionality of the shingled disassembler is to eliminate bytes that are clearly data (because all flows that contain them lead to execution of bytes that do not encode any valid instruction), and to compose a byte sequence that retains information for generating every possible valid shingle of the source binary. This is a major benefit of this approach since the shingled disassembly encodes a superset of all the possible valid disassemblies of the binary. In later sections, we discuss how we apply our graph disassembler to prune this superset until we find the most probable byte classifications. In order to define what consists of a valid execution path, we must first discuss a few key concepts.

Definition 2. ***Fall through***[1]
Shingle x (conditionally) falls through *to shingle y, denoted $x \rightharpoondown y$, if shingle y is located adjacent to and after instruction x, and the semantics of instruction x do not (always) modify the program counter. In this case, execution of instruction x is (sometimes) followed by execution of instruction y at runtime.*

Definition 3. ***Unconditional Branch***
A shingle is an unconditional branch *if it only falls through when its operand explicitly targets the immediately following byte. Unconditional branch instructions for x86 include `jmp` and `ret` instructions.*

Unconditional branch instructions are important in defining valid disassemblies because the last instruction in any disassembly must be an unconditional branch. If this is not the case, the program could execute past the end of its virtual address space.

Definition 4. ***Static Successor***
A control-flow edge (x,y) is static *if $x \rightharpoondown y$ holds or if x is a conditional or unconditional branch with fixed (i.e., non-computed) destination y. An instruction's* static successors *are defined by $S(x) = \{y \mid (x,y) \text{ is static}\}$.*

Definition 5. ***Postdominating Set***
The (static) postdominating set *$P(x)$ of shingle x is the transitive closure of S on $\{x\}$. If there exists a static control-flow from x to an illegal address (e.g., an address outside the address space or whose bytes do not encode a legal instruction), then $P(x)$ is not well defined and we write $P(x) = \perp$.*

[1]At first glance, it would seem that we could strengthen our defintion of fall-throughs to any two instructions that do not have an unconditional branch instruction between them. However, there are cases where a compiler will place a `call` and `jcc` instruction followed by data bytes. A common example of this is `call [IAT:ExceptionHandler]` since the exception handler function will never return.

Definition 6. ***Valid Execution Path***
All paths in $P(x)$ are considered valid execution paths *from x.*

The x86 instruction set does not make use of every possible opcode sequence; therefore certain bytes cannot be the beginning of a code instruction. For example, the `0xFF` byte is used to distinguish the beginning of one 7 different instructions, using the byte that follows to distinguish which instruction is intended. However, `0xFFFF` is an invalid opcode that is unused in the instruction set. This sequence of bytes is common because any negative offset in two's complement that branches less than `0xFFFF` bytes away starts with `0xFFFF`. The shingled disassembler can immediately mark any shingle whose opcode is not supported under the x86 instruction set as *data*. A shingle that is marked as *data* is either used as the operand of another instruction, or it is part of a data block within the code section. Execution of the instruction would cause the program to crash.

Lemma 1. ***Invalid Fall-through***
$\langle \forall x, y :: x \rightharpoondown y \wedge y := \emptyset \rightarrow x := \emptyset \rangle$*, in which $\emptyset$ stands for data bytes.*

Any time that we encounter an address that is marked data, all fall-throughs to that instruction can be marked as data as well. Direct branches also fall into this definition. All direct `call` and `jmp` instructions imply a direct executional relationship between the instruction and its target. Therefore, any shingle that targets a shingle previously marked as data is also marked as data.

Definition 7. ***Sheering***
A shingle x is sheered *from the shingled disassembly when $\forall y :: x \rightharpoondown y$, x and all y are marked as data in the shingled disassembly.*

Figure 2 illustrates how our shingled disassembler works. Given a binary of byte sequence `6A 01 51 56 8B C7 E8 B6 E6 FF FF` ..., the shingled disassembler performs a single-pass, ordered scan over the byte sequence. Data bytes and invalid shingles are marked along the way. Figure 2(a) demonstrates the first series of valid shingles, beginning at the first byte of the binary. Figure 2(b) starts at the second byte, which falls through to a previously disassembled shingle. The shingle with byte `C7` is then marked as data (shaded in Figure 2(c)) since it is an invalid opcode. Figure 2(d) shows an invalid shingle since it falls through to an invalid opcode `FF FF`. Our shingled disassembler marks the two shingles `B6` and `FF` as invalid in the sequence. Figure 2(e) shows another valid shingle that begins at the ninth byte of the binary. After completing the scan, our shingled disassembler has stored information necessary to produce all valid paths in $P(x)$.

The secondary function of the shingled disassembler is to collect local statistics called code/data modifiers that are specific to the executable. These modifiers keep

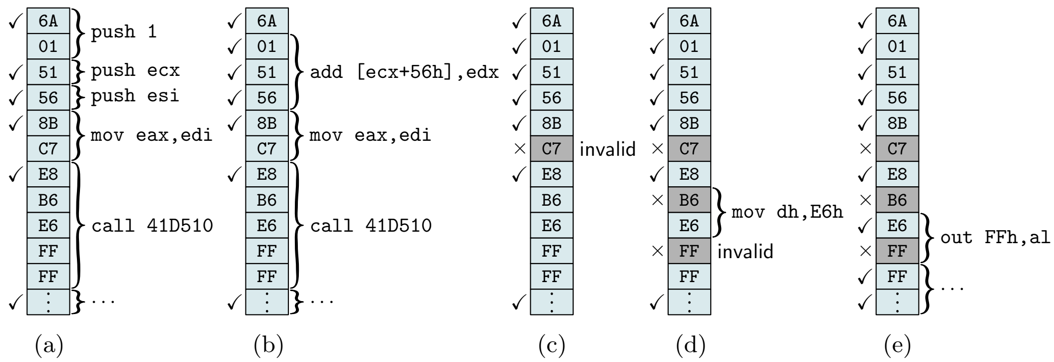

Figure 2: Shingled disassembly of a sample byte sequence: (a) a shingle sequence beginning at the first byte; (b) a shingle sequence beginning at the second byte; (c) a non-shingle that starts with an invalid opcode; (d) a shingle that falls through to an invalid opcode; and (e) a shingle sequence beginning at the ninth byte.

track of the likelihood that a shingle is code or data in this particular executable. The following heuristics are used to update modifiers:

1. If the shingle at address a is a long direct branch instruction with a' as its target, the address a' is more likely to be a code instruction. We apply this heuristic with short direct branches as well, but with less weight since two byte instructions are more likely to be seen within other instruction operands.

2. If three shingles sequentially fall-through to each other and match one the most common instruction opcode sequences, each of these three addresses is more likely to be code. Common sequences include function prologues, epilogues, etc.

3. If bytes at address a and $a + 4$ both encode addresses that reference shingles within the code section of the binary, the likelihood that addresses a through $a + 7$ are data is very high. Shingles a through $a + 7$ are marked as data, as well as any following four byte sequences that match this criteria. This is most likely a series of addresses referenced by a conditional branch elsewhere in the code section.

The pseudocode for generating a shingled disassembly for a binary is shown in Figure 3. For simplicity, the heuristics used to update modifiers are not described in the pseudocode. Lines 1–17 construct a static control-flow graph G in which all edges are reversed. A distinguished node `bad` is introduced with outgoing edges to all shingles that do not encode any valid instruction, or that branch to static,

Input: $x_0, \ldots, x_{n-1} \in [0, 2^8)$
Output: $y_0, \ldots, y_{n-1} \in \{\mathtt{data}, \mathtt{maybe\_code}\}$

```
1   G := ∅
2   for a := 0 to n − 1 do
3     y_a := maybe_code
4     i := decode(x_a x_{a+1} ···)
5     if i is undefined then
6       G.insert(bad, a)
7     else
8       if i falls through then
9         if a + |i| < n then G.insert(a + |i|, a)
10        else G.insert(bad, a)
11      endif
12      if i is a static jump/branch then
13        if is_exec_ok(dest(i)) then G.insert(dest(i), a)
14        else G.insert(bad, a)
15      endif
16    endif
17  endfor
18  foreach a ∈ depth_first_search(G, bad) do
19    y_a := data
20  endfor
```

Figure 3: Shingled disassembly algorithm

non-executable addresses. Lines 18–20 then mark all addresses reachable from `bad` as data. The rest are possible instruction starting points.

3.2 Opcode State Machine

The state machine is constructed from a large corpus of pre-tagged binaries, disassembled with IDA Pro v6.3. The byte sequences of the training executables are used to build an opcode graph, consisting of opcode states and transitions from one state to another. For each opcode state, we label its transition with the probability of seeing the next opcode in the training instruction streams. The opcode graph is a probabilistic finite state machine (FSM) that encodes all the correct disassemblies of the training byte sequences annotated with transition probabilities. The accepting state of the FSM is the last unconditional branch seen in the binary.

Figure 4 shows what this transition graph might look like if the x86 instruction set only contained four opcodes: 0x01 through 0x04. Each directed edge in the graph between opcode x_i and x_j implies that a transition between x_i and x_j has been observed in the corpus, and the edge weight of $x_i \to x_j$ is the probability that given

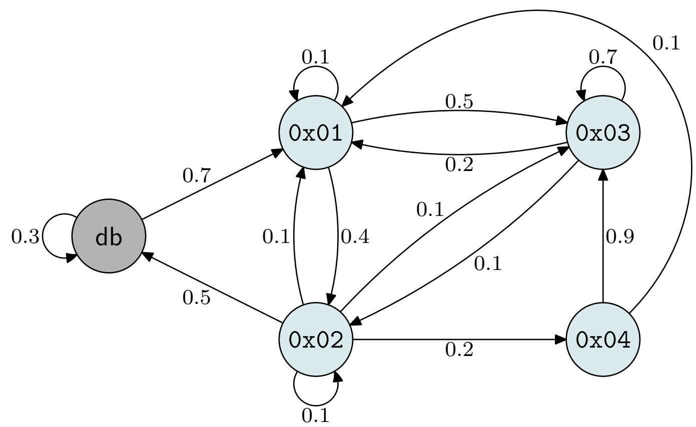

Figure 4: Instruction transition graph: 4 opcodes

x_i, the next instruction is x_j. It is also important to note the node *db* in the graph which represents data bytes. Any transition from an instruction to data observed in the corpus will be represented by a directed edge to the *db* node. The graph for the full x86 instruction set includes more than 500 nodes, as each observed opcode must be included.

3.3 Maximum-Likelihood Execution Path

We name the output of the shingled disassembler a *shingled binary.* The shingled binary of the source executable encodes within it up to 2^n possible valid disassemblies. Our graph disassembler is designed to scan the shingled binary and prune shingles with lower probabilities. By using our graph disassembler, we can find the maximum-likelihood set of byte classifications by tracing the shingled binary through the opcode finite state machine. At every receiving state, we check which preceding path (predecessor) has the highest transition probability. For example in Figure 2, the 5th byte (`8B`) is the receiving state of two preceding addresses: byte 1 (see Figure 2(a)) and byte 2 (see Figure 2(b)). We compute the transition probability from each of the two addresses and sheer the one with a lower probability.

Theorem 1. *The graph disassembler always returns the maximum-likelihood byte classifications among the set $\mathcal{S}$ of all valid shingles.*

Proof. Each byte in the shingled binary is a potential receiving state of multiple predecessors. At each receiving state, we keep the best predecessor with the highest transition probability. Therefore, when we reach the last receiving state—the accepting state, which represents the last unconditional brach instruction—we find the shingle with the highest probability as the best execution path. $\square$

The transition probability of a predecessor consists of two parts: the global transition probability taken from the opcode state machine and the local modifiers, and local statistics of each byte being code or data based on several heuristics. This is important because runtime reference patterns specific to the binary being disassembled are included in distinguishing the most probable disassembly path.

Let r be a receiving state of a transition triggered at x_i in the shingled binary, let $Pr(pred(x_i))$ be the transition probability of the best predecessor of x_i, and let cm and dm be the code and data modifiers computed during shingled disassembly. The transition probability to r is as follows:

$$Pr(r) = Pr(pred(x_i)) * cm/dm$$

if x_i is a fall-through instruction, or

$$Pr(r) = Pr(pred(x_i)) * cm/dm * Pr(db_i) * Pr(db_r)$$

if x_i is a branch instruction, where $Pr(db_i)$ is the probability that x_i is followed by data and $Pr(db_r)$ is the probability that r is proceeded by data. Every branch instruction can possibly be followed by data. To account for this, when determining the best predecessor for each instruction, branch instructions are treated as fall-throughs to their following instruction and to data. Each branch instruction can be a predecessor to the following instruction or to any instruction that is on a 4-byte boundary and is reachable via data bytes.

Therefore, the transition probability of any valid shingle-path s resulting in a trace of $r_0, \ldots, r_i, \ldots, r_k$ is:

$$Pr(s) = Pr(r_0)Pr(r_1) \cdots Pr(r_i) \cdots Pr(r_k)$$

and the optimal execution path s^* is:

$$s^* = \arg\max_{s \in \mathcal{S}} Pr(s).$$

3.4 Algorithm Analysis

Our disassembly algorithm is much quicker than other approaches of comparable accuracy due to the small amount of information that needs to be analyzed. The time complexity of each of the three steps is as follows:

- Shingled disassembly: Lines 1–17 of Figure 3 complete in $O(n)$ time (where n is the number of bytes in executable sections) and construct a CFG G with at most $2n$ edges. The depth-first search in Lines 18–20 is linear in the size of G. We conclude that the algorithm in Figure 3 is $O(n)$.

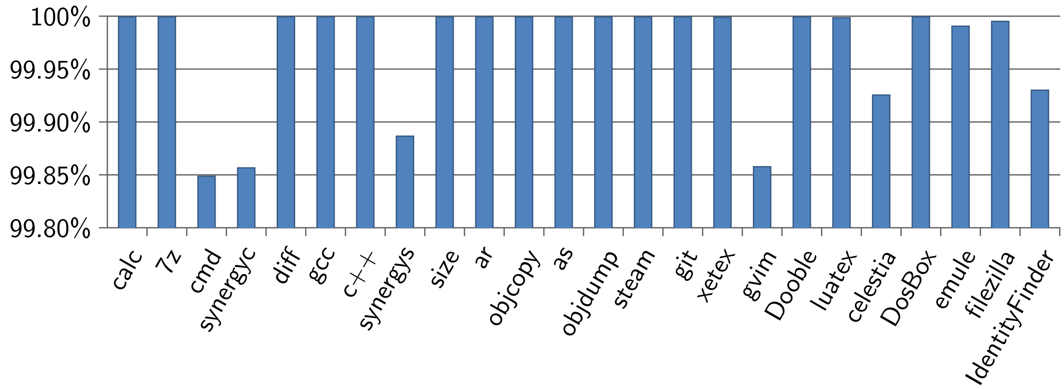

Figure 5: Percent of instructions identified by IDA Pro that were also identified by our disassembler

- Sheering: Pruning invalid shingles also requires $O(n)$ time.
- Graph disassembly: The graph-based disassembler performs a single-pass scan over the shingled binary, and is therefore also $O(n)$.

Therefore, our disassembly algorithm runs in time $O(n)$, that is, linear in the size of the source binary executable.

4 Evaluation

A prototype of our shingled disassembler was developed in Windows using Microsoft .NET C#. Testing of our disassembly algorithm was performed on an Intel Xeon processor with six 2.4GHz cores and 24GB of physical RAM. We tested 24 difficult binaries with very positive results.

4.1 Broad Results

Table 1 shows the different programs on which we tested our disassembler, as well as file sizes and code section sizes. It also displays the number of instructions that the graph disassembler identified that IDA Pro didn't identify as code. Figure 5 shows the percentage of instructions that IDA Pro identified as code that our disassembler also identified as code.

After the shingled disassembly has been composed, each binary has already had a large number of instructions eliminated as invalid opcodes or invalid fall-throughs. Figure 6 shows the percentage of bytes that have been sheered after the shingled disassembly.

Table 1: File Statistics

File Name	File Size (KB)	Code Size (KB)	# Instr. Missed by IDA
calc	114	75	1700
7z	163	126	680
cmd	389	129	5449
synergyc	609	218	12607
diff	1161	228	3002
gcc	1378	254	2760
c++	1380	256	2769
synergys	738	319	8061
size	1703	581	5540
ar	1726	593	8626
objcopy	1868	701	6293
as	2188	772	7463
objdump	2247	780	7159
steam	1353	860	16928
git	1159	947	9776
xetex	14424	1277	18579
gvim	1997	1666	19145
Dooble	2579	1884	57598
luatex	3514	2118	18381
celestia	2844	2136	24950
DosBox	3727	3013	24217
emule	5758	3264	52434
filezilla	7994	7085	79367
IdentityFinder	23874	12781	180176

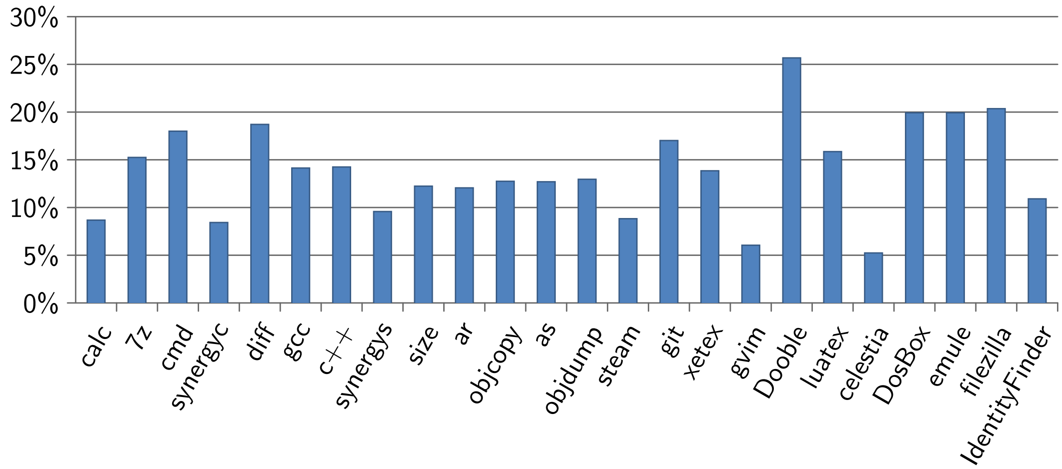

Figure 6: Percent of addresses sheered during shingled disassembly

Our disassembler runs in linear time in the size of the input binary. Figure 7 shows how many times longer IDA Pro took to disassemble each binary relative to our disassembler. Our disassembler is increasingly faster than IDA Pro as the size of the input grows.

Finally, for each binary we used Ollydbg to create and save the traces of executions. Tracing executions in this way does not reveal the ground truth of non-executed bytes (which may be data or code), but the bytes that do execute are definitely code. We compared these results to the static disassembly yielded by our disassembler, by IDA Pro, and by the dynamic disassembly tool VDB/Vivisect [4]. Figure 8 shows the results. Both our disassembler and IDA Pro were 100% accurate against the execution paths that actually executed during the tests, but VDB/Vivisect exhibited much lower accuracies of around 15–35%. We also used VDB/Vivisect to dynamically trace command line tools, such as the Spec2000 benchmark suite and Cygwin, and obtained similar code coverages. This provides significant evidence that purely dynamic disassembly is not a viable solution to many disassembly problems where high code coverage is essential.

4.2 eMule Case Study

The eMule file sharing software is very popular, with almost five hundred million downloads on SourceForge. It also works extremely well as a case study to compare our disassembler versus IDA Pro to examine some of the mistakes that IDA Pro makes.

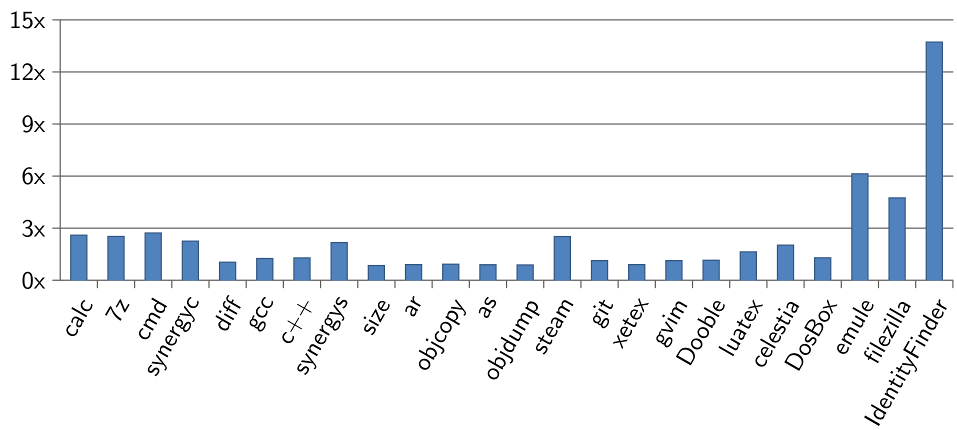

Figure 7: Ratio of IDA Pro's disassembly time to our disassembly time

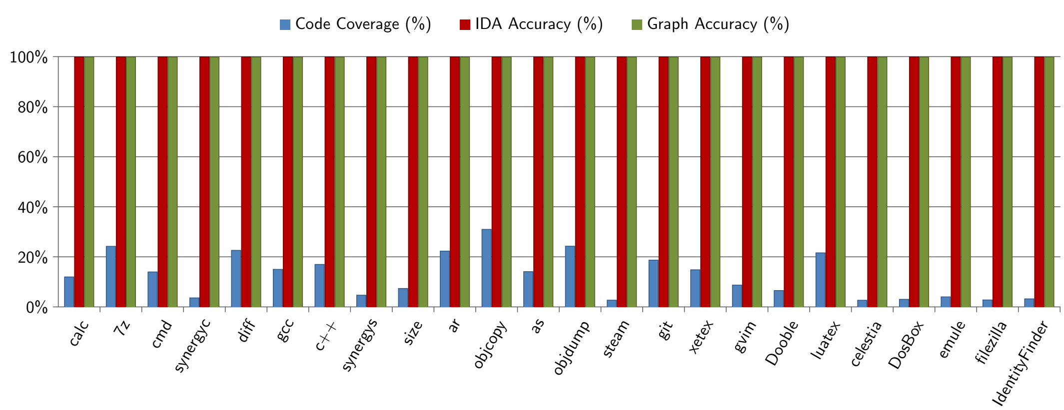

Figure 8: Coverage of observed execution traces by IDA Pro, VDB/Vivisect, and our disassembler

We tested IDA Pro v6.3 against our disassembler when working with eMule v.50a. IDA Pro makes a large number of mistakes when attempting to disassemble eMule and ignores vast blocks of code. Our disassembler does not make these mistakes.

The most pervasive mistake made by IDA Pro is demonstrated in Cases 1–2, where a large block of instructions follows a call that IDA's heuristics infer to be non-returning. IDA therefore misclassifies the bytes as data and fails to include the instructions they encode in its disassembly. We observed this error at at least nine other addresses (`0x524CF0`, `0x5250A0`, `0x525C00`, `0x5262D3`, `0x533090`, `0x62ABBB`, `0x6B2821`, `0x6CF68A`, and `0x711DC9`). More examples of this may exist in eMule; these are merely the instances that were manually verified by the authors. Our disassembler accurately classifies each of these blocks as code.

IDA Pro sometimes drops a single common first byte from an instruction. In Case 3, `0x8B` is dropped, and `0xFF` is dropped in Case 4. This is an obvious mistake since IDA's disassembly implies that code falls through to data. Our disassembler is incapable of making this mistake due to its shingling disassembly algorithm.

Cases 5, 6 and 7 are all very similar, each demonstrating IDA Pro's susceptibility to dropping direct branch instructions. Each of these instructions should be classified as code; for example, in Case 7 the `jmp` instruction implements a switch statement. Our disassembler correctly identifies and classifies all of these instructions.

Finally, Case 8 exhibits an entire function epilogue that IDA misclassifies as data. Function epilogues are among the most common opcode sequences seen in binaries, so IDA's misclassification is surprising. We speculate that it may arise from an undesirable interaction between two or more of its many heuristics, one of which made a misclassification that overrode the rest. Our disassembler correctly classifies the bytes since our state machine recognizes the high likelihood of this sequence appearing as code.

5 Conclusion

We presented an extremely simple yet highly effective static disassembly technique using probabilistic finite state machines. It finds the most probable set of byte classifications from all possible valid disassemblies. Compared to the current state-of-the-art IDA Pro, our disassembler runs in time linear in the size of the input binary. We achieve greater efficiency, and experiments indicate that our resulting disassemblies are more accurate than those yielded by IDA Pro.

We are currently working on extending our disassembler to instrument and record the actual execution traces of executables, for better estimation of ground truth and therefore more comprehensive evaluation of accuracy. One major challenge is to get high code coverage—the percentage of the code sections covered during

Table 2: Disassembly Comparison for *emule.exe*

IDA Pro	**Ours**
Case 1: Missed code after a possibly non-returning call (158-1104 bytes)	
41CF9D: call CxxThrowException@8	41CF9D: call CxxThrowException@8
41CFA2: dw 0CC5Bh	41CFA2: pop ebx
...	41CFA3: db align (0CCh x13)
41D030: dd 0CC5B0028h, (0CCh x12)	41CFB0: push 0FFFFFFFFh
41D040: push 0FFFFFFFFh	...
	41D032: pop ebx
	41D033: db align (0CCh x13)
	41D040: push 0FFFFFFFFh
Case 2: Missed code after a possibly non-returning call (26 bytes)	
41CD3D: call CxxThrowException@8	41CD3D: call CxxThrowException@8
41CD42: db '[?????????????d',0	41CD42: db 5B
41CD53: align 4	41CD43: db align (0CCh x13)
41CD54: dd 548B0000h, 0FF6A0824h	41CD50: mov eax, large fs:0
41CD5C: push offset SEH_41CC30	41CD56: mov edx, [esp+8]
	41CD5A: push 0FFFFFFFFh
	41CD5C: push offset SEH_41CC30
Case 3: 0x8B byte dropped (1 byte)	
525D82: mov edx, [eax+1Ch]	525D82: mov edx, [eax+1Ch]
525D85: db 8Bh	525D85: mov edi, off_7DFAB4
525D86: cmp eax, offset off_7DFAB4	
Case 4: 0xFF byte dropped (1 byte)	
58DC4E: push ecx	58DC4E: push ecx
58DC4F: db 0FFh	58DC4F: call off_7DFAB4
58DC50: adc eax, offset off_7DFAB4	
Case 5: Short direct jmp dropped (2 bytes)	
67882D: call sub_6C978E	67882D: call sub_6C978E
678832: db 0EBh	678832: jmp short loc_678839
678833: db 5	678834: cmp ebp, 0FFFFFFFEh
678834: cmp ebp, 0FFFFFFFEh	
Case 6: Long direct jump dropped (5 bytes)	
71951C: mov ecx, [ebp-10h]	71951C: mov ecx, [ebp-10h]
71951F: db 0E9h	71951F: jmp CWnd@@UAE@XZ
719520: dd 0FFFBA052h	
Case 7: Dropped jump switch statement (14 bytes)	
6C3137: db 83h	6C3137: sub esp, 2Ch
6C3138: dd 0E0832CECh, 8524FF3Fh	6C313A: and eax, 3Fh
6C3140: dd offset off_7DF12E	6C313D: jmp off_7DF12E[eax*4]
6C3144: fdiv st, st	6C3144: fdiv st, st
Case 8: Dropped epilogue (6 bytes)	
6CF231: db 59h	6CF231: pop ecx
6CF232: db 5Fh	6CF232: pop edi
6CF233: db 5Eh	6CF233: pop esi
6CF234: db 0C2h	6CF234: retn 8
6CF235: db 8	
6CF236: db 0	

each execution—especially for large applications. The instrumented execution traces would give us the advantage to verify all identified code sections in a controlled and automatic fashion.

Acknowledgments

The research reported herein was supported in part by NSF award #1054629, AFOSR award FA9550-10-1-0088, and ARO award W911NF-12-1-0558. Any opinions, recommendations, or conclusions expressed are those of the authors and do not necessarily reflect those of the NSF, AFOSR, or ARO.

References

[1] C. Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler.* No Starch Press, Inc., San Francisco, California, 2008.

[2] Hex-Rays. The IDA Pro disassembler and debugger. `www.hex-rays.com/idapro`.

[3] Intel. Intel® architecture software developer's manual. `http://www.intel.com/design/intarch/manuals/243191.htm`, 2011.

[4] Invisigoth of KenShoto. Visipedia. `http://visi.kenshoto.com`.

[5] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, pages 423–427, 2008.

[6] J. Kinder and H. Veith. Precise static analysis of untrusted driver binaries. In *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 43–50, 2010.

[7] J. Kinder, F. Zuleger, and H. Veith. 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)*, pages 214–228, 2009.

[8] N. Krishnamoorthy, S. Debray, and K. Fligg. Static detection of disassembly errors. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE)*, pages 259–268, 2009.

[9] G. Project. Gnu binary utilities. `http://sourceware.org/binutils/docs-2.22/binutils/index.html`, 2012.

[10] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, pages 45–54, 2002.

[11] E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, and R. Carrasco. Probabilistic finite-state machines – part I. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(7):1013–1025, 2005.

[12] E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, and R. Carrasco. Probabilistic finite-state machines – part II. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(7):1026–1039, 2005.

[13] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham. Differentiating code from data in x86 binaries. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, volume 3, pages 522–536, 2011.