Probabilistic Disassembly

Kenneth Miller*, Yonghwi Kwon†, Yi Sun*, Zhuo Zhang*, Xiangyu Zhang*, Zhiqiang Lin‡

*Department of Computer Science, Purdue University, West Lafayette, USA
†Department of Computer Science, University of Virginia, Charlottesville, USA
‡Department of Computer Science and Engineering, Ohio State University, Columbus, USA

Abstract—Disassembling stripped binaries is a prominent challenge for binary analysis, due to the interleaving of code segments and data, and the difficulties of resolving control transfer targets of indirect calls and jumps. As a result, most existing disassemblers have both false positives (FP) and false negatives (FN). We observe that uncertainty is inevitable in disassembly due to the information loss during compilation and code generation. Therefore, we propose to model such uncertainty using probabilities and propose a novel disassembly technique, which computes a probability for each address in the code space, indicating its likelihood of being a true positive instruction. The probability is computed from a set of features that are reachable to an address, including control flow and data flow features. Our experiments with more than two billions binaries show that our technique does not have any FN and has only 3.7% FP. In comparison, a state-of-the-art superset disassembly technique has 85% FP. A rewriter built on our disassembly can generate binaries that are only half of the size of those by superset disassembly and run 3% faster. While many widely-used disassemblers such as IDA and BAP suffer from missing function entries, our experiment also shows that even without any function entry information, our disassembler can still achieve 0 FN and 6.8% FP.

I. INTRODUCTION

Analyzing and transforming commercial-off-the-shelf and legacy software have many applications [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], such as bug finding, security hardening, reverse engineering, code clone detection and refactoring. However, they are highly challenging due to the lack of source code. The first fundamental problem is to precisely disassemble the software. The seemingly simple task is indeed highly challenging due to the diversity and complexity of compilation and optimizations. There are two popular kinds of disassembly techniques. The first one disassembles instructions following the address order, called linear sweep disassemblers, and the other disassembles instructions by following control flow edges (e.g., jumps and calls), called traversal disassemblers. Both have well known limitations. In particular, code and data can interleave, causing a large number of false positives and even false negatives in linear sweep disassemblers; traversal disassemblers suffer indirect control flow caused by function pointers, virtual tables, and switch-case statements, which make recognizing control transfer targets highly difficult. Even the state-of-the-art disassemblers such as those in BAP [22], IDA-Pro [23], OllyDbg [24], Jakstab [25], SecondWrite [26], and Dyninst [27] have difficulty fully disassembling complex binaries [28]. Some may miss up to 30% of the code [28]. There are machine learning based methods [29] that aim to recognize function entries by instruction patterns (e.g., starting with “push ebp”). However, such methods have inevitable false positives and false negatives (e.g., the entries of many library functions do not follow specific patterns). Recently, superset disassembly [30] was proposed to address these limitations. It disassembles at each address to produce a superset of instructions. A rewriter is built on the disassembler to instrument all superset instructions. While it has a critical guarantee of no false negatives that other binary rewriting tools cannot provide, the rewritten binaries have substantial code size blow-up and nontrivial runtime overhead (e.g., 763% size overhead and 3% runtime overhead on SPEC programs).

We argue that the capabilities of reasoning about uncertainty is critical for binary analysis, since it is inherent due to the lack of symbolic information. Our overarching idea is hence to use probabilities to model uncertainty and then perform probabilistic inference to determine the appropriate way of disassembling subject binaries. In particular, our disassembler computes a posterior probability for each address in the code section to indicate the likelihood of the address denoting a true positive instruction (i.e., an instruction generated by the compiler). Specifically, our technique disassembles the binary at each address just like superset disassembly. We call the result the superset instructions or valid instructions, which may or may not be true positives. We then identify correlations between these superset instructions such as one being the transfer target of another; and one defining a register that is later accessed by another. These relations denote semantic features that only the real code body would likely demonstrate. We call them hints. They are uncertain because instructions decoded from random bytes may by chance possess such features. For each kind of hint, we perform apriori probability analysis to determine their prior probabilities. We develop an algorithm to aggregate these hints and compute the posterior probabilities. The resulting disassembler has probabilistic guarantees of no false negatives (e.g., the likelihood of missing a true positive instruction is lower than \( \frac{1}{1000} \)). In our empirical study with 2,064 binaries, it never misses any true positive instruction with an appropriate setting. It also has a much smaller number of false positives and much lower overhead in rewriting, compared with superset disassembly.

Our contributions are summarized as follows.

- We propose an innovative idea of probabilistic disassembling. The capabilities of reasoning about uncertainty provides unique benefits compared to existing techniques.
- We identify a set of features for use as disassembly hints.
<table>
<thead>
<tr>
<th>Disassembler</th>
<th>False Negative</th>
<th>False Positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear sweep</td>
<td>Some</td>
<td>Substantial</td>
</tr>
<tr>
<td>Traversal [23]</td>
<td>Substantial</td>
<td>None</td>
</tr>
<tr>
<td>Superset [30]</td>
<td>None</td>
<td>Bloated</td>
</tr>
<tr>
<td>Our method</td>
<td>None*</td>
<td>Some*</td>
</tr>
</tbody>
</table>

*: with probabilistic guarantees

TABLE I: Comparison of Different Kinds of Disassemblers and perform static probability analysis to determine their likelihood (§III-B).

- We develop a novel inference algorithm that leverages a number of key characteristics of x86 instruction design (§IV) to aggregate uncertain hints.
- Our experiments on 2,064 binaries demonstrate that our technique does not have any false negatives, and the false positive rate is 3.7%, meaning that it disassembles 3.7% additional instructions that are not true positives. It does not miss any instructions even when function entries are not available, with 6.8% FP. Our evaluation on SPEC Windows PE binaries shows that objdump misses 3095 instructions due to code and data interleavings, whereas our tool misses none with 8.12% FP. We also use our disassembler in supporting binary writing. When compared with the state-of-the-art superset rewriting technique [30], our technique reduces the size of rewritten binary by about 47% and improves the runtime speed of the rewritten binary by 3%.

II. BACKGROUND AND MOTIVATION

In this section, we use a real world example to explain binary code disassembly, the limitations of existing work (§II-A), and how we advance the state of the art (§II-B).

A. Binary Code Disassembly

Figure 1(a) presents a snippet from libUbuntuCompo-

nents.so in Ubuntu 16.04. In this piece of code, data is inserted in between the code bodies of two functions. In (a), the bytes from 0xbbf72 to 0xbbf8f (in blue) denote data. Address 0xbbf90 denotes the entry of a function. Another function (omitted from the figure) precedes the data bytes. While the binary is stripped, we acquire the ground truth through debug symbols from a separate unstripped instance.

Linear Sweep Disassembly. Linear sweep disassemblers disassemble the next instruction from the bytes right after the current instruction. Here, we use objdump. Without symbolic information, objdump cannot recognize the data bytes. As a result, after it disassembles the body of the preceding function, it proceeds to disassemble the data bytes to instructions 0xbbf72, 0xbbf8b, and so on as in Figure 1(b). Specifically, in the shaded area, it considers the three bytes starting at 0xbbf8f an instruction. Consequently, it misses the true function entry 0xbbf90. Note that the instruction sequences in Figure 1 are horizontally aligned by their addresses. In addition, objdump disassembles the wrong instruction at 0xbbf92. This illustrates that linear disassemblers cannot properly handle inter-leavings of data and instructions. Note that embedding data such as constant values and jump tables in between code segments is a common practice in compilers [28], [31]. As presented in Table I, linear sweep disassemblers have some false negatives (i.e., missing instructions) and a lot of false positives (i.e., incorrectly disassembling data bytes as instructions). False negatives are particularly problematic for binary rewriting as missing even a single instruction could have catastrophic consequences. False positives can cause unnecessary overhead in rewriting, ambiguity in type reverse engineering and so on.

Traversal based Disassembly. Some other dissemblers such as IDA [23] and BAP [22] disassemble by following control flow edges, starting from function entries. A prominent challenge is to recognize function entries. Missing an entry means the entire function body may not be properly disassembled. The presence of indirect calls makes function entry identification difficult as the precise call targets are only known at runtime. In our example, there is no direct invocation to the function entry 0xbbf90 in libUbuntuComponent and the function is not exported either. As a result, IDA misses the entire function body. Furthermore, the first instruction of the function entry is a rarely used instruction “MOV 0x19b978(rip), rax”. As such, ML based techniques (e.g., [32], [29], [33]) likely miss it. There are also non-learning techniques to recognize functions in binaries [34], [35], [36]. They are based on heuristics such as the matching of push and pop operations at the entry and exit of a function. However, a systematic way to handle the inherent uncertainty in such heuristics is still in need.

As illustrated by Table I, traversal disassemblers have no false positives but potentially substantial false negatives. In fact, Bao et al. [29] show that traversal disassemblers such as IDA may miss 68.19% function entries.

Superset Disassembly. A state-of-the-art technique (particularly for rewriting/instrumentation) is called superset disassembly [30]. The idea is to consider that every address starts an instruction, called superset instruction. As such, consecutive superset instructions may share common bytes. Rewriting is performed on all superset instructions. It can be easily inferred that the superset disassembler has no false negatives but must have a bloated code body due to the large number of superset instructions that are not true positives (Table I). Figure 1(c) presents the results for superset disassembly. Observe that a superset instruction is generated by disassembling the bytes starting at each address. Hence, we have instructions at 0xbbf72, 0xbbf73, ..., 0xbbf91, 0xbbf92, and so on. Observe that consecutive instructions share common byte values (e.g., the body of 0xbbf91 “8b 05 71 b9 19’00” is the suffix of 0xbbf90). Also observe that all the true positive instructions, i.e., those in Figure 1(a), are part of the superset. As such, the rewritten binary can properly execute as all possible jump/call targets must be instructions in the superset and hence instrumented. Note that the bloated instructions cause not only substantial size overhead, but also runtime slowdown because executing each
superset instruction requires a table lookup to determine the location of the instrumented version.

**B. Our Technique**

We aim to inherit the advantages of superset disassembly (i.e., no false negatives) while substantially reducing the false positives and achieving much lower overhead. The idea is that true positives have lots of hints indicating that they are true instructions. For example, they often have a lot of definition and use (def-use) relations caused by registers and memory, that is, a register/memory-location is defined at an earlier instruction and then used in a later one. In Figure 1(a), hint 1 indicates a def-use relation caused by register rax between instructions 0xbbf90 and 0xbbfa2; 2 by rdx; 3 indicates a def-use by the flag bit. Note that false positive instructions are less likely to induce def-use relations due to their random nature. For example, instructions at 0xbbfba (highlighted ones) have large probabilities (some of them are almost certain such as 0xbbf0 and 0xbbfa), whereas false positives have (very) small probabilities.

![Fig. 2: Occlusion does not cascade](image)

**III. PROBABILISTIC CHARACTERISTICS OF X86**

**A. Observing Instruction Occlusion**

In x86, part of a valid instruction may be another valid instruction and two valid instructions may have overlapping
bodies. We call them occluded instructions. We say a few bytes form a valid instruction if they can be decoded to an instruction. A valid instruction may not be a true positive instruction. Therefore, if the starting point (e.g., function entry) is not properly recognized, we may have an occluded instruction sequence that differs from the true positive sequence.

Consider an example in Figure 2. Column one shows the continuous addresses; column two shows the byte values; and the remaining columns show different instructions sequences when disassembling starts at different addresses. Note that each instruction (box) aligns horizontally with its addresses and byte values in the first two columns. Column three shows the ground truth instruction sequence, in which the first four bytes (from 0x400597 to 0x40059a) form a MOV instruction whereas the following five bytes form another MOV instruction, followed by a CALL instruction. However, if we start disassembling in the middle of the first instruction, we could acquire sequences of valid instructions that occlude with the ground truth, as shown in the remaining columns (i.e., occluded instructions are in grey). Observe that in columns four and five, part of the MOV instruction is decoded to a different MOV instruction and a conditional jump instruction, respectively. In the last column, the last byte 0xe0 of the MOV instruction even groups with the first byte 0xbf of the next (ground truth) instruction to form a valid LOOPNE instruction.

A concern about occlusion is that it may be cascading, meaning that when we start at a wrong place, a large number of following instructions are consequently occluded. However, researchers have the following observation [37].

(Occlusion Rule): Cascading occlusion is highly unlikely: occluded sequences tend to quickly agree on a common suffix of instructions.

If one of the sequences is the true positive sequence, occluded sequences quickly converge with the true positive. Consider the example in Figure 2. The three occluded sequences all converge to the ground truth sequence after one or two instructions. Intuitively, cascading occlusion is unlikely because: two occluded instructions have a good chance to agree on their rears. In other words, the suffix of an instruction is likely to be another instruction. Consider Figure 2. The occluded instructions in columns 3 and 4 are the suffixes of the ground truth MOV instruction. The only exception is that when an occluded instruction \( i_0 \) (e.g., the LOOPNE instruction in Figure 2 last column) starts at the very end of a valid instruction \( j_0 \) (e.g., the first MOV in the 3rd column), \( i_0 \) may go beyond \( j_0 \) and cause occlusion in the instruction following \( j_0 \), say \( j_1 \) (e.g., the second MOV in the 3rd column). In this case, \( i_0 \) likely ends in the middle of \( j_1 \). As such, the instruction(s) following \( i_0 \) (e.g., the SUB and ADD instructions in the last column) agree with \( j_1 \) at the their rear ends. We did a study on 2064 ELF binaries and found that 99.992% occluded instruction sequences converge within four instructions. We have also conducted a formal probability proof from the encodings of x86 instructions. Our proof shows that for instructions \( i_0, \ldots, i_k \) with \( n_0, \ldots, n_k \) bytes, respectively. The probability of an occluded sequence starting inside \( i_0 \) and not agreeing with the rear of \( i_k \) is at most \( \frac{1}{(n_0-2) \cdots (n_k-2)} \). With a sequence of 7 instructions, each having 5 bytes, the probability that an occluded sequence does not converge at all is \( \frac{1}{3} = \frac{1}{6561} \). Intuitively, it is analogous to that if two parties cannot settle on a dispute with a small probability \( p \) in one round of negotiation. The probability that they cannot resolve within \( n \) rounds is \( p^n \). The details are elided.

B. Observing Probabilistic Hints for Disassembling

Without knowing the appropriate entries of code segments, we could disassemble at each address and acquire a set of all valid instructions (or, superset instructions [30]) with only some being true positives. Next we discuss a number of correlations between valid instructions that indicate that the corresponding bytes are not data bytes with high probabilities. We call them probabilistic hints. The occlusion rule and the probabilistic hints are the two corner stones of our technique.

Hint I: Control Flow Convergence. As shown in the middle of Figure 3 (b), if there are three potential instructions \( instr_1, instr_2 \) and \( instr_3 \) with \( instr_3 \) being the transfer target of both \( instr_1 \) and \( instr_2 \), there is a good chance that they are not data bytes (but rather instruction bytes). Figure 3(a) shows an example. The bytes starting at 0x804a634 and at 0x804a646 are disassembled to two conditional jumps \( \mathcal{A} \) and \( \mathcal{B} \), respectively, whose target is a same valid instruction \( \mathcal{C} \). Intuitively, since it is highly unlikely data bytes can form two control transfer instructions and both by chance point to the same target, they are likely instruction bytes. This control flow relation is often induced by high level language structures such as conditional statements (e.g., Figure 3(c)).

Probability Analysis. Assume data byte values have uniform distribution. Given two valid control transfer instructions \( instr_1 \) and \( instr_2 \), let \( instr_1 \)'s transfer target be \( t \), which has the range of \([-2^{17}+1, 2^{17}-1], [-2^{15}+1, 2^{15}-1], \text{and} [-2^{31}+1, 2^{31}-1] \) for relative, near, and long jumps, respectively. The likelihood that \( instr_2 \) has the same transfer target is hence \( \frac{1}{2^{17}}, \frac{1}{2^{31}} \), and \( \frac{1}{2^{31}} \). In other words, when we see two control transfer instructions having the same target, the likelihood that they are data bytes is (very) low.

Hint II: Control Flow Crossing. As shown in the middle of Figure 4 (b), if there are three valid instructions \( instr_1, instr_2 \) and \( instr_3 \), with \( instr_2 \) and \( instr_3 \) next to each other; \( instr_3 \) being the transfer target of \( instr_1 \), and \( instr_2 \) having
a control transfer target different from instr3 (and hence crossing control flow edges), there is a good chance that they are not data bytes (but rather instruction bytes). Figure 4 (a) shows an example. Since it is highly unlikely data bytes can form two control transfer instructions with one jumping to right after the other, they are likely instructions. This control flow relation is often induced by loopy language structures (e.g., Figure 4 (c) with instr1 the loop head, instr2 the last instruction of the loop body and instr3 the loop exit). The probability analysis is similar to that of control flow convergence and hence elided.

There are also other control flow related hints. For example, if a valid control transfer instruction i (e.g., a jump) has a target that does not occlude with the sequence starting from i, the chance of i denoting data bytes is \(\frac{n}{n+1}\), with n the average instruction length. This is because a false positive jump (disassembled from random data bytes) may likely jump to the middle of an instruction. Although this hint is not as strong as the convergence and crossing hints, a large number of such hints can be aggregated to form strong indication, through an algorithm described in §IV.

### Hint III: Register Define-use Relation

We say a pair of instructions instr1 and instr2 have a register define-use (def-use) relation, if instr1 defines the value of a register (or some flag bit) and instr2 uses the register (or the flag bit). In Figure 5(c), there are two def-use relations denoted by the arrows, one induced by register rdx and the other by eax. Another example is that a flag bit is set by a comparison instruction, and then used by a following conditional jump instruction. Given two valid instructions, if they have def-use relation, they are unlikely data bytes.

Note that false positive instructions often do not have register def-use although they may demonstrate (bogus) memory def-use relations. Figure 5(a) presents a snippet of jump table disassembled to a sequence of instructions. Observe that the first instruction adds al to the memory location indicated by rax whereas the second instruction adds cl to the same location. There is a memory def-use between the two instructions as the second instruction first reads the value stored in the location and then performs the addition. However, as we will show in later probability analysis, register def-use is hardly random, but rather caused by register allocation (by compiler). Figure 5(b) presents a snippet of string. It is disassembled to a sequence of valid instructions too. Observe that there are no register def-use relations.

### Probability Analysis

Assume data byte values have uniform distribution. To simplify our discussion, we further assume an arbitrary valid instruction has \(\frac{1}{4}\) chance to write to some register or some flag bit (and the other \(\frac{1}{4}\) chance writing to only memory). In contrast, an arbitrary valid instruction reading some register is much more likely. Note that even a read from memory often entails reading from register. For example, the instruction at 0x4005ce in Figure 5(c) performs a memory read which entails reading rbp. Hence, we make an approximation (just for the sake of demonstrating our probability analysis), assuming the likelihood that an instruction reads some register is 0.99. Each instruction has three bits to indicate which register is being read/re-writen-to according to the x86 instruction reference. As such, given two valid instructions instr1 and instr2, they have register def-use with the chance of \(\frac{3}{2} \times \frac{1}{2} = \frac{3}{16}\). In other words, when we observe def-use between two valid instructions, the chance that they denote data bytes is \(\frac{1}{16}\).

We need to point out these hints only indicate the corresponding bytes are not data bytes, they do not suggest the valid instructions are indeed true positives. In other words, they may be occluded instructions that are part of some ground truth instructions. This is because occluded instructions often share similar features such as the same register operand(s). For instance, bytes “89 c2”, which is the suffix of the first instruction in Figure 5 (c), is disassembled to MOV eax, edx, which also has a register def-use with the second instruction. However, observing these hints strongly suggests that the corresponding bytes are instruction bytes. Fortunately, the aforementioned occlusion rule dictates that even there is occlusion, it will soon be automatically corrected. Our disassembly technique is hence built on this observation.

Besides the register def-use hint, we have other hints that denote data flow related program semantics. For example, an instruction saving a register to a memory location followed by another instruction that defines the register corresponds to register spilling [38], which can hardly be random. We also consider memory def-use between instructions of different opcodes. Details are elided.

### IV. Probabilistic Disassembling Algorithm

As discussed in the previous section, when a probabilistic hint is observed, we have certain confidence that the corresponding bytes are not data bytes but rather instruction bytes, although we are still uncertain if they are true positive instructions as their occluded peers may have similar properties as well. The occlusion rule dictates that a sequence that starts with some occluded instruction can quickly correct itself and converge on true positive instructions. Therefore in our method, we consider an instruction is likely a true positive if multiple sequences with a large number of hints converge on the instruction. Here, a sequence starting from an instruction i is acquired by following the control flow (e.g., if i is an unconditional jump, the next instruction in the sequence would be the target of the jump). We say multiple sequences converge on an instruction if it occurs in all of them.

Specifically, let a hint h have a prior probability p being data byte, with p computed by the analysis in the previous section.
Fig. 5: Register Definition-use Relation

Fig. 6: Example for the algorithm; the code snippet in \texttt{foo()} corresponds to a statement “for (i=0; i<11; i++) ...”

Since the following instructions are acquired strictly following the control flow semantics, they inherit the probability \( p \).

Intuitively, if \( j \) is the next instruction of \( h \) along control flow, \( j \)’s probability of being some data byte is equal to or smaller than \( p \). When the sequences starting with multiple hints \( h_1, h_2, \ldots, h_n \) converge on an instruction \( i \), the probability of \( i \) representing a data byte is \( D[i] = p_1 \times p_2 \times \cdots \times p_n \). As such, when a large number of hints converge on \( i \), \( i \) is highly unlikely a data byte.

However, a small \( D[i] \) does not necessarily denote that \( i \) is a true positive instruction. We then leverage the exclusion property of a true positive instruction, that is, if \( i \) is a true positive instruction, all the other valid instructions occluding with \( i \) must not be true positive instructions. Therefore, we compute the likelihood of \( i \) being a true positive instruction by conducting normalization with all the instructions occluded with \( i \). Intuitively, if \( i \) is the only one that has a very small \( D[i] \) compared to all the occluded instructions, \( i \) is highly likely true positive. If there are occluded instructions whose \( D \) values are comparable to \( D[i] \), we cannot be certain that \( i \) is true positive. In this case, we keep all these instructions just like superset disassembly. However, due to the loopy structures in binary, the algorithm is overall iterative, and terminates when a fix point is reached. The iterative analysis is in lines 8-30 with variable \texttt{fixed} point used to determine termination. The analysis consists of three steps: forward propagation of hints (lines 10-21), local propagation within occlusion space (lines 22-24), and backward propagation of invalidity (lines 25-30). The first step traverses from the beginning of \( B \) to the end, propagating/collecting hints and computing the aggregated probabilities. It leverages the following forward inference: (1) the control flow successor of a (likely) instruction is also a (likely) instruction. Otherwise, the program is invalid because its execution would lead to exception (caused by the invalid instruction) following the control flow. The second step is to propagate the computed probability for each instruction \( i \) to its occlusion space consisting of all the other addresses that can be decoded into 6.

Algorithm Details. Algorithm 1 takes as input a binary \( B \) which is an array of bytes indexed by address; a list of hints \( H \) with \( H[i] = p \) meaning that \( i \) is a hint with a prior probability \( p \) (of being data bytes). It produces posterior probabilities \( P \) with \( P[i] \) the likelihood that \( i \) being a true positive instruction.

Within the algorithm, we use \( D[i] \) to denote the probability \( i \) being a data byte and \( RH[i] \) to denote the set of hints that reach \( i \), each hint represented by its address. In lines 1-6, the algorithm initializes all the \( D \) values and all the \( RH \) values. If the bytes starting at \( i \) denote invalid instruction, \( D[i] \) is set to 1.0, otherwise \( \perp \) to denote that we do not have any knowledge. Note that some byte sequences cannot be disassembled to any valid instruction.

In lines 1-6, the algorithm initializes all the \( D \) values and all the \( RH \) values. If the bytes starting at \( i \) denote invalid instruction, \( D[i] \) is set to 1.0, otherwise \( \perp \) to denote that we do not have any knowledge. Note that some byte sequences cannot be disassembled to any valid instruction.

1This property may not hold in manually crafted binaries in which the developer purposely introduces occlusion between true positive instructions. However, we focus on binaries generated by compilers in this paper.
instructions occluding with $i$. It is to leverage the following local inference: (2) an instruction being likely renders all the other instructions in its occlusion space unlikely. The third step traverses each address from the end to the beginning and propagates invalidity of instructions. It leverages the following backward inference: (3) when an instruction $i$ is unlikely, all the instructions that reach $i$ through control flow are unlikely. Intuitively, it is the logical contrapositive of the forward inference rule (1). The first step can be considered to identify instruction bytes, whereas the second and third steps are to identify data bytes.

**Step I.** In lines 13-15, if $i$ denotes a hint and $i$ has not been added to $RH[i]$, it is added to $RH[i]$ and $D[i]$ is updated to the product of the prior probabilities of all the hints in $RH[i]$ (line 15). In lines 16-21, the algorithm propagates the hints in $RH[i]$ to $i$'s control flow successor(s). Particularly, if $RH[i]$ has some hint that the successor $n$ does not have (line 17), the hints of $i$ are propagated to $RH[n]$ by a union operation (line 18), and $D[n]$ is updated. In lines 20-21, if the successor $n$ has a smaller address so that it has been traversed in the current round, the analysis needs another round to further propagate the newly identified hint(s).

**Step II.** In lines 22-24, the algorithm traverses all the addresses and performs local propagation of probabilities within occlusion space of individual instructions. Particularly, for each address $i$, it finds its occluded peer $j$ that has the minimal probability (i.e., the most likely instruction). The likelihood of $i$ being data is hence computed as $1 - D[j]$ (line 24).

**Step III.** Lines 25-30 traverse from the end to the beginning. For each address $i$, if its control flow predecessor $p$ does not have any computed probability or has a smaller probability (line 27), which intuitively means that we have more evidence that $i$ is data (instead of instruction), then we set $p$ to have the same level of confidence of denoting data bytes (line 28). In the extremal case, if $D[i] \equiv 1.0, D[p]$ must be 1.0 too. If $p$ has a larger address than $i$ and hence $p$ must have been traversed, variable $fixed_point$ is reset and the analysis will be conducted for another round (lines 29-30).

Note that the control flow successors and predecessors are implicitly computed along the analysis. Our analysis does not require correctly recognizing indirect jump and target calls, which is a very difficult challenge. In other words, even though such control flow relations are missing, our technique can still collect enough hints from (disconnected) code blocks to disassemble correctly. In §V-D, we show that our technique can disassemble without any function entry information with 0 false negatives and only 6.8% false positives.

After the iterative process, lines 31-38 compute the posterior probabilities for true positive instructions by normalization. If an instruction starting at $i$ is invalid, $P[i]$ is set to 0 (lines 32-33). Otherwise, it sums up the inverse of probability $D$ for all the instructions occluded with $i$, including $i$ itself, to $s$; then $P[i]$ is computed as the ratio between $\frac{1}{D[i]}$ and $s$.

**Example.** Consider an example in Figure 6. It is much simpler than the one in §II and allows easy explanation. The large box on the left shows a code snippet denoting the beginning of a function `foo()` (from 0x40058c to 0x400594) and part of the function body (from 0x40055c0 to 0x4005e1) corresponding to a simple loop “for (i=0;i<11;i++)...”. The code snippet is preceded by data bytes that stand for constant strings (from 0x400057b to 0x40058b). The strings are disassembled to valid instructions. Note that symbolic information is not available, we mark the function entry and strings just for explanation purpose. Boxes $A$-$F$ on the right stand for sequences starting from some occluded instructions. The instructions in the grey background denote occlusions whereas instructions without background denote the converged ones, which are horizontally aligned with the

<table>
<thead>
<tr>
<th>Algorithm 1 Probabilistic Disassembling</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> $B$ - binary indexed by address</td>
</tr>
<tr>
<td><strong>Output:</strong> $P[i]$ - posterior probability of an address $i$ denoting a true positive instruction</td>
</tr>
<tr>
<td><strong>Variable:</strong> $D[i]$ - posterior probability of an address $i$ being data byte</td>
</tr>
</tbody>
</table>

1: for each address $i$ in $B$ do
2: if invalidInstr($i$) then
3: $D[i] \leftarrow 1.0$
4: else
5: $D[i] \leftarrow \perp$
6: $RH[i] \leftarrow \emptyset$
7: $fixed_point \leftarrow false$
8: while $!fixed_point$ do
9: $fixed_point \leftarrow true$
10: for each address $i$ from start of $B$ to end do
11: continue
12: if $H[i] \equiv \perp$ and $i \notin RH[i]$ then
13: $RH[i] \leftarrow RH[i] \cup \{i\}$
14: $D[i] \leftarrow \Pi_{h \in RH[i]} H[h]$
15: for each $n$, the next instruction of $i$ along control flow do
16: if $RH[i] - RH[n] \neq \emptyset$ then
17: $RH[n] \leftarrow RH[n] \cup RH[i]$
18: $D[n] \leftarrow \Pi_{h \in RH[n]} H[h]$
19: if $n < i$ then
20: $fixed_point \leftarrow false$
21: $fixed_point \leftarrow false$
22: for each address $i$ from start of $B$ to end do
23: if $D[i] \equiv \perp$ and $\exists j$ occluding with $i$, s.t. $D[j] \neq \perp$ then
24: $D[i] \leftarrow 1 - min_j \{ \text{occludes with } D[j] \}$
25: for each address $i$ from end of $B$ to start do
26: if $D[i] \equiv \perp$ or $D[p] < D[i]$ then
27: $D[p] \leftarrow D[i]$
28: if $p > i$ then
29: $fixed_point \leftarrow false$
30: for each address $i$ from start of $B$ to end do
31: if $D[i] \equiv 1.0$ then
32: $P[i] \leftarrow 0$
33: continue
34: $s \leftarrow \frac{1}{D[i]}$
35: for each address $j$, representing an instruction occluded with $i$ do
36: $s \leftarrow s + \frac{1}{D[j]}$
37: $P[i] \leftarrow \frac{1}{D[i]}$
corresponding instructions in the leftmost box. For example, in box A, disassembling at 0x40057c causes occlusion up to 0x400583. In the following, we show how our algorithm computes the probabilities for true positives.

During preprocessing, our technique collects the hints and their prior probabilities. Each circled number denotes such a hint (only part of the hints are shown). For example, (1) is a register-def-use hint (hint III in §III-B) due to rd1. According to §III-B, the prior probability is \( \frac{1}{16} \) (being a data byte). Note that this hint actually occurs in the data bytes. In addition, (2) and (3) stand for the register-spilling (i.e., backup and then update) hint due to rbp and rbp, respectively; (4) stands for register-def-use; (5) stands for control-flow-crossing (hint II in §III-B); and (6) stands for memory-def-use. None of the occluded sequences provide any additional hints.

Initially, \( D[0x400583] = D[0x40057e] = 1.0 \) and all other \( D \) values are \( \perp \). In step I, hints are collected and probabilities are computed in a forward fashion. Hint (1) cannot be propagated to address 0x400584 due to the bad instruction at 0x400583 and the sequences in A and D do not provide any hint, hence \( D[0x400584] = \perp \). Its occluded peers in 0x400585-0x400588 have the same \( D \) value.

In contrast, \( D[0x40058c] = \frac{1}{16} \) due to the hint (2). Similarly, \( D[0x40058d] = \left( \frac{\perp}{16} \right)^3 \) due to the three hints it is involved in. As shown in boxes B, its occluded peer 0x40058e cannot be reached from 0x40058c. As a result, it gets no hint and \( D[0x40058c] = \perp \). Similarly \( D[0x40058f] = \perp \). Let us skip a few instructions and consider 0x4005db. Due to the loop (with the backedge 0x4005df \( \rightarrow \) 0x4005c2), hints (2)-(6) all reach 0x4005db. As such, \( D[0x4005db] \) is a tiny value smaller than \( \frac{1}{16} \). In contrast, as shown in C and F, no hints can reach its occluded peers 0x4005dc and 0x4005dd and their \( D \) values remain \( \perp \). Through step II of local propagation in occlusion space, \( D[0x40058f] = D[0x40058e] = 1 - \frac{1}{16} \) and \( D[0x4005db] = D[0x4005dd] \approx 1 \).

In step III, the invalidity information is propagated backward. That is, if an address has a larger \( D \) value than its predecessor, the predecessor inherits that \( D \) value. Specifically, 0x400583 being invalid invalidates all its control flow predecessors including 0x400582, 0x400581, 0x40057f, and 0x40057b. That is, their \( D \) values equal to 1.0.

In contrast, 0x40058d has two possible predecessors, “0x40058b: 4c 55 rex.WR PUSH rbp” (not shown in the code snippet) and “0x40058c: 55 PUSH rbp” (shown in the code snippet). The former has the prefix “rex” that is only used in the long mode [39] and hence does not form any hint with other instructions. Furthermore, it occludes with 0x40058c. As a result, \( D[0x40058b] = 1 - D[0x40058c] = \frac{15}{16} \) after steps I and II. However, since \( D[0x40058d] = \frac{1}{16} \), which is smaller than \( D[0x40058b] \), there is no backward propagation. Although \( D[0x40058c] = 1 - \frac{1}{16} \) is a large value, it does not have any control flow predecessor, that is, it cannot be reached by disassembling at any preceding addresses.

After the iterative process, the \( D \) values are normalized to compute the posterior probabilities.

For example, since \( 0x40058c \) only occludes with \( 0x40058b \) and \( D[0x40058b] = \frac{15}{16} \), \( D[0x40058c] = \frac{1}{16} \). \( P[0x40058c] = \frac{16}{17} \times \frac{15}{16} = 0.94 \) and \( P[0x40058b] = 0.058 \). The other true positive instructions have higher than 0.99 probabilities. For instance, \( P[0x40058d] \approx 0.9987 \) and \( P[0x40058e] = P[0x40058f] \approx 0.0006 \). \( P[0x4005db] \approx 1.0 \) and \( P[0x4005dd] \) are negligible.

V. IMPLEMENTATION AND EVALUATION

We have implemented a prototype on top of BAP [22] using OCaml. Our implementation has 5,546 LOC. To evaluate our technique, we use two sets of benchmarks. The first set contains 2,064 x86 ELF binaries collected from the BAP corpora [22]. The size of these binaries ranges from 100KB to 3MB. They come with symbolic information, from which we derive the ground truth. We stripped the binaries before applying our disassembler. The second set is the SPEC2006INT programs. We used SPEC for the comparison with super set disassembly [30]. All the experiments were run on a machine with Intel i7 CPU and 16 GB RAM. Our evaluation addresses the following research questions (RQ).

- **RQ1:** Can our technique disassemble binaries with accuracy, completeness, and efficiency (§V-A)?
- **RQ2:** How does our technique compare with a state of the art super set disassembly (§V-B)?
- **RQ3:** How does our technique perform when data and code are interleaved, in comparison with linear sweep disassembly (§V-C)?
- **RQ4:** How does our technique perform when no function entry information is available (e.g., for indirect functions that are one of the most difficult challenges for traversal disassemblers in IDA [23] and BAP [22]) (§V-D)?

A. RQ1: Effectiveness and Efficiency

To answer RQ1, we perform four experiments: (1) measure false negatives (missing true positive instructions) and false positives (bogus instructions) on the 2,064 binaries; (2) measure the disassembling time; (3) analyze the contributions of each individual kind of hints; (4) study the effect of different probability threshold settings.

**FP and FN.** We report the results with the probability threshold of \( P \geq 0.01 \), meaning that we are very conservative and hence keep all the valid instructions with more than 0.01 computed posterior probability. In this setting, our technique does not have any false negatives. Figure 7 shows the correlations between binary size and the FP rate. Observe that most cases cluster at bottom-left. Most medium to large binaries have lower than 5% false positives. The a few largest (on the right) are even lower than 2%. The ones with larger FP rates tend to be small binaries, which have fewer hints. The average FP rate is only 3.7%. This strongly suggests the effectiveness of our technique.

**Disassembling Time.** Figure 8 shows the distribution of time. Observe that it has a close-to-linear relation with the binary size. The largest ones take about 10 minutes to disassemble.
The medium ones take 4-8 minutes. Our algorithm is not as fast as other disassemblers because it is an iterative algorithm based on probabilistic inference. Also, we have not optimized implementation. We argue that since disassembling is one-time effort, the cost is justifiable.

Contributions of Different Kinds of Hints. Figure 9 shows the results for three settings: using only the control flow hints; only the data flow hints (e.g., def-use and register-spilling); and using all hints. The $x$ axis denotes intervals of the FP rate and the $y$ axis represents the number of binaries that fall into an interval. For example, with only control flow hints, about 300 binaries have less than 1% FPs; with only data flow hints, about 70 binaries have less than 1% FPs; with all hints, the number is 510. In other words, both types of hints are critical for getting the best results.

Effects of Different Probability Thresholds. As mentioned earlier, we retain instructions whose computed probability $P > \alpha$. Figure 10 shows how the FP, FN rates (on the right $y$ axis) and the percentage of precisely disassembled functions (on the left $y$ axis) change with $\alpha$ (the $x$ axis). For example, at the starting point on the left is $\alpha = 0.67\%$ (i.e., we keep instructions with $P > = 0.0007$), FP is about 4% and FN is 0, and 53.23% of the 607,758 functions in the corpora are precisely disassembled. With the growth of $\alpha$, FP drops, FN and the rate of precisely disassembled functions rise. At the other end on the right is $\alpha = 20\%$, FP is 0.6% whereas FN is 6.7%. Almost 73% of functions are precise.

B. RQ2: Comparison with Superset Disassembly

Linear sweep and traversal disassemblers suffer false negatives, which may cause serious problems in binary rewriting. Superset disassembler [30] is a state-of-the-art that does not have false negatives. However, it introduces lots of false positives, leading to size blowup in rewriting and unnecessary runtime overhead. Table II shows the comparison with superset disassembly. To compare the effects on binary rewriting, we integrate our disassembler with their rewriter. We use the same SPEC programs in [30] (column one). Columns 2-4 present the FP rate, the code size blowup after rewriting, and the execution time variation after rewriting, respectively. Here, we do not add any instructions during rewriting. Columns 5-7 present the same information for our technique. Observe that we reduce the size blowup from 763% to 404% and improve the execution time by 3%. Note that it is normal that rewritten binaries may execute faster than the original code [30] due to the cache behavior changes caused by rewriting. Note that although our technique still has 404% size inflation, it is because the rewriter uses a huge lookup table to translate each address in the code space. While all the entries are necessary in superset rewriting, majority of these entries are not needed in our rewriter, and therefore empty. We plan to remove the empty table entries and replace it with a cost-effective hash table in the future. The FP rate differences (columns 2 and 5) indicate the large number of these redundant entries.

C. RQ3: Handling Data and Code Interleavings

A prominent challenge in disassembly is to handle data and code interleavings (i.e., the presence of read-only data in between code segments), which could cause false negatives.
In this section, we discuss related work. Probabilistic inference has been used in program analysis, such as locating software faults [40], inferring explicit information flow [41], and recognizing memory objects [42]. But to our best knowledge, we are the first one to use it in binary disassembly. Machine learning has been used for binary analysis. For instance, Wartell et. al. [43] used a statistical compression technique to differentiate code and data. Shingled Graph Disassembly [44] leverages graph model based learning on a large corpus of binaries to recognize data bytes. Our technique does not require training. Its formalization of using a random variable to represent each address, the introduction of hints and the fusion of these hints are unique. Dynamic disassembly (e.g., [45], [46], [27], [1], [47]) disassembles during execution. These approaches impose extra runtime overhead. In addition, they can hardly serve downstream static analysis such as dependence analysis. Disassembly has many applications, such as binary hardening [6], [48], [49], [50], [5], deobfuscation [51], [52], reassemble disassembling [53], [54], [55], reverse engineering [56], and exploitation [57]. Our work is particularly suited in rewriting and hardening.

Table II: Superset Disassembly vs Probabilistic Disassembly

<table>
<thead>
<tr>
<th>Program</th>
<th>PP</th>
<th>Size (rewritten/orig)</th>
<th>Exec. time (rewritten/orig)</th>
<th>PP</th>
<th>Size (rewritten/orig)</th>
<th>Exec. time (rewritten/orig)</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.per bench</td>
<td>85.32%</td>
<td>780%</td>
<td>116.71%</td>
<td>11.29%</td>
<td>427%</td>
<td>117.74%</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>84.65%</td>
<td>779%</td>
<td>105.49%</td>
<td>6.57%</td>
<td>400%</td>
<td>97.30%</td>
</tr>
<tr>
<td>403.gcc</td>
<td>88.03%</td>
<td>751%</td>
<td>104.60%</td>
<td>11.33%</td>
<td>409%</td>
<td>101.71%</td>
</tr>
<tr>
<td>429.mc</td>
<td>84.72%</td>
<td>749%</td>
<td>104.02%</td>
<td>4.60%</td>
<td>399%</td>
<td>104.74%</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>90.27%</td>
<td>727%</td>
<td>103.43%</td>
<td>6.20%</td>
<td>372%</td>
<td>97.30%</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>82.71%</td>
<td>779%</td>
<td>99.14%</td>
<td>6.64%</td>
<td>411%</td>
<td>94.12%</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>87.08%</td>
<td>756%</td>
<td>98.83%</td>
<td>7.61%</td>
<td>407%</td>
<td>92.76%</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>80.96%</td>
<td>758%</td>
<td>100.42%</td>
<td>4.04%</td>
<td>400%</td>
<td>96.94%</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>82.36%</td>
<td>781%</td>
<td>100.39%</td>
<td>2.41%</td>
<td>395%</td>
<td>94.57%</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>85.02%</td>
<td>768%</td>
<td>105.24%</td>
<td>9.82%</td>
<td>420%</td>
<td>108.4%</td>
</tr>
<tr>
<td>473.astar</td>
<td>81.46%</td>
<td>761%</td>
<td>94.28%</td>
<td>3.90%</td>
<td>402%</td>
<td>93.24%</td>
</tr>
<tr>
<td>Avg</td>
<td>84.8%</td>
<td>763%</td>
<td>103.0%</td>
<td>6.8%</td>
<td>404%</td>
<td>99.9%</td>
</tr>
</tbody>
</table>

![Graph Disassembly](image)

Fig. 11: FP Rates In the Absence of Function Entries in linear sweep disassembly. In this experiment, we compile SPECint 2000 benchmark by Visual Studio 2017 with different optimization levels to generate a set of binaries. We extract ground truth from pdb files. We use both objdump, a linear sweep disassembler, and our disassembler to disassemble the stripped binaries. The comparison between the disassembled results and the ground truth shows that objdump misses 3095 instructions in total, whereas our tool misses none. The average FP rate of our tool is 8.12% (5.95%, 8.84%, 5.76%, and 8.98% for optimization levels O1, O2, Od, and Ox, respectively). The FP rate is higher than ELF binaries as data and code interleavings are more common in PE binaries.

D. RQ4: Handling Missing Function Entries

Another prominent challenge, especially for traversal disassembly, is missing function entries due to indirect calls. To simulate such challenges, we eliminate all the function related hints, such as call edges that have the same target (part of the hint 1). In other words, we only leverage the intra-procedural hints to disassemble. Figure 11 presents the results, with x axis the FP interval and y axis the number of binaries. The average FP rate is 6.8%, slightly higher than that of using both inter- and intra-procedural hints. FN is still 0. This indicates that in the cases where traversal disassemblers such as IDA and BAP have troubles due to missing function entries, our technique has substantial advantages.

VI. RELATED WORK

We have discussed existing disassembly techniques in §II. In this section, we discuss other related works. Probabilistic inference has been used in program analysis, such as locating software faults [40], inferring explicit information flow [41], and recognizing memory objects [42]. But to our best knowledge, we are the first one to use it in binary disassembly. Machine learning has been used for binary analysis. For instance, Wartell et. al. [43] used a statistical compression technique to differentiate code and data. Shingled Graph Disassembly [44] leverages graph model based learning on a large corpus of binaries to recognize data bytes. Our technique does not require training. Its formalization of using a random variable to represent each address, the introduction of hints and the fusion of these hints are unique. Dynamic disassembly (e.g., [45], [46], [27], [1], [47]) disassembles during execution. These approaches impose extra runtime overhead. In addition, they can hardly serve downstream static analysis such as dependence analysis. Disassembly has many applications, such as binary hardening [6], [48], [49], [50], [5], deobfuscation [51], [52], reassemble disassembling [53], [54], [55], reverse engineering [56], and exploitation [57]. Our work is particularly suited in rewriting and hardening.

VII. THREATS TO VALIDITY

Although we used the corpus from BAP and SPEC in our experiments, the benchmarks may not represent all features of real-world binaries. We will test our technique on more binaries. We focus on binaries generated by compilers. It is unclear how our technique will perform on obfuscated code although we believe semantic hints still exist in such code.

VIII. CONCLUSION

We propose a novel probabilistic disassembling technique that can properly model the uncertainty in binary analysis. It computes a probability for each address in the code space, indicating the likelihood of the address representing a true positive instruction. The probability is computed by fusing a set of uncertain features that can reach the address. The results show that our technique produce no false negatives and as low as 3.7% false positives; and it substantially outperforms a state-of-the-art superset disassembly technique.

ACKNOWLEDGMENT

The authors were supported in part by ONR N0001414104068, N000141712947, and N000141712995, DARPA FA8650-15-C-7562, NSF 1748764, 1409668, 1834213, and 1834215, AFOSR FA95501410119, and Sandia National Lab under award 1701331.
REFERENCES