Probabilistic Disassembly

***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. We therefore 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 features that are reachable to an address, including control flow and data flow features. Our experiments with more than two thousands binaries show that our technique does not have any FN and 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 of them 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 rewriting tool is built on top of 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 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.

Disassembler	False Negative	False Positive
Linear sweep	Some	Substantial
Traversal [23]	Substantial	None
Superset [30]	None	Bloated
Our method	None*	Some*
*: with probabilistic guarantees		

TABLE I: Comparison of Different Kinds of Disassemblers

- Our prototype 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. 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 `libUbuntuComponents.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 diassemblers 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

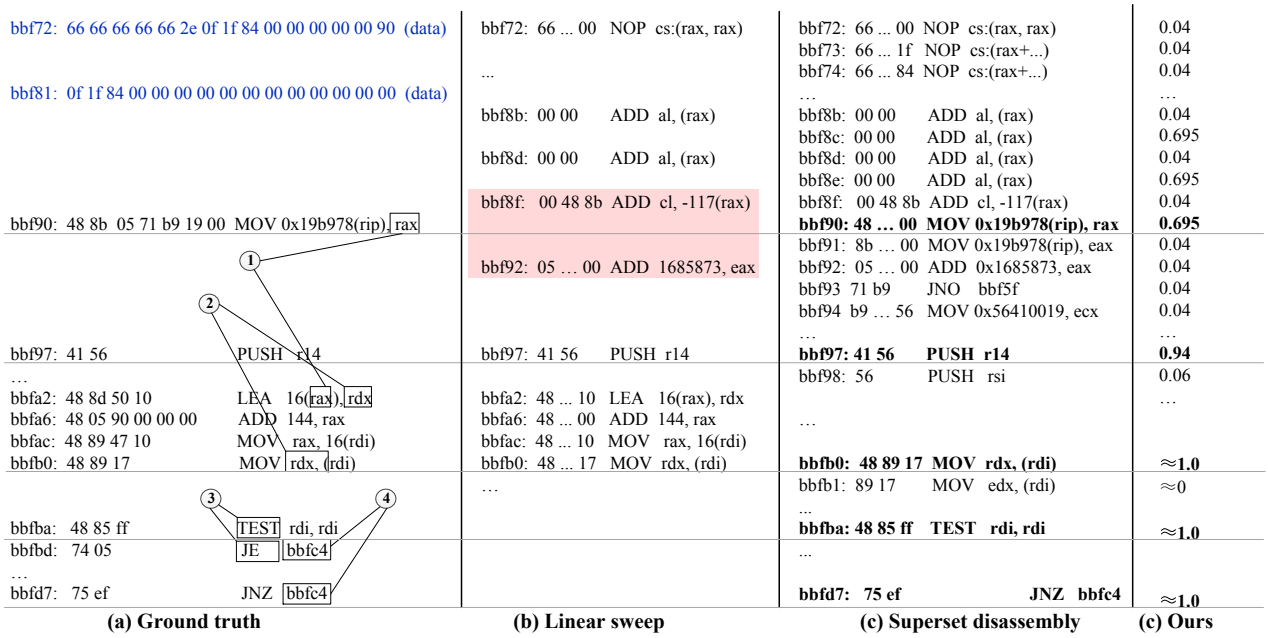

Fig. 1: Example from libUbuntuComponent.so. Instructions are horizontally aligned by their addresses. The code is slightly modified for demonstration purposes. In instructions with two operands, the first one is source and the second one is destination.

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 `0xbbf8b`-`0xbbf8f` (Figure 1(c)) define some memory indexed by `rax`, but there are no corresponding uses. Furthermore, two jumps to the same target are likely true positives (e.g., hint ④) as the chance that random jumps have the same target is small. More hints are discussed in §III-B.

However, hints are uncertain, meaning that false positives instructions have a (small) chance of exhibiting such features. For example, according to §III-B, false positive instructions may have $\frac{1}{16}$ chance to have def-use relation caused by some register. Hence, *the essence of our technique is to associate these hints with prior probabilities that are derived from apriori probability analysis, and then perform probabilistic inference to fuse these evidences to form strong confidence about true positives*. Intuitively, the inference procedure that aggregates prior probabilities is based on the following reasoning: *if a superset instruction is likely to be a true positive, its control flow descendants are likely to be true positives, and the different superset instructions that share common bytes with it are unlikely true positives*. Note that we aim to disassemble binaries generated by regular compilers so that instructions do not have overlapping bodies. For example, the instructions involved in hints ①-④ have reachability along control flow (e.g., those in ① can reach ④), allowing their probabilities to be progagated and aggregated. Intuitively, while individually ①-④ have certain probability (e.g., $\frac{1}{16}$) to be random, the chance of all of them randomly happening together is very low. After inference, the posterior probabilities indicate the likelihood of superset instructions being true positives. Figure 1(d) shows the probabilities computed by our technique for each superset instruction. Observe that the true positives (highlighted ones) have large probabilities (some of them are almost certain such as `0xbbfb0` and `0xbbfba`), whereas false positives have (very) small probabilities.

Addr	Value	Ground Truth	Start @400598	Start @400599	Start @40059a
400597	48	MOV rsi, -0x20(rbp)			
400598	89		MOV esi, -0x20(rbp)		
400599	75			JNE 400599	
40059a	e0				LOOPNE 0xff...ffc1
40059b	bf	MOV 0x28, edi	MOV 0x28, edi	MOV 0x28, edi	
40059c	28				SUB al, (rax)
40059d	00				
40059e	00				ADD al, (rax)
40059f	00				
4005a0	e8	CALL 400468	CALL 400468	CALL 400468	CALL 400468
…	...				

Fig. 2: Occlusion does not cascade

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 suffices 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, ..., i_k with n_0, ..., 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 $\frac{1}{(n_0-2)\ldots(n_k-2)}$. With a sequence of 7 instructions, each having 5 bytes, the probability that an occluded sequence does not converge at all is $\frac{1}{3^7} = \frac{1}{6561}$. 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.

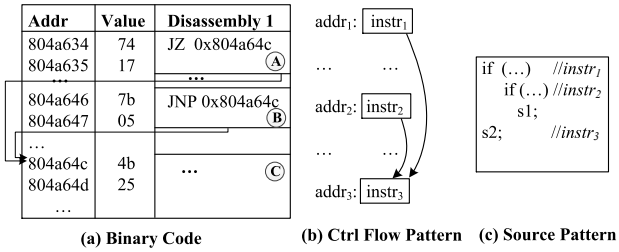

Fig. 3: Control flow convergence

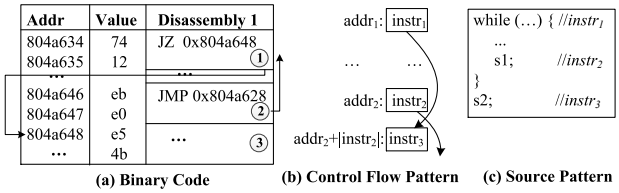

Fig. 4: *Control Flow Crossing*

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 Ⓐ and Ⓑ, respectively, whose target is a same valid instruction Ⓒ. 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 [-2^7+1, 2^7-1], [$-2^{15}+1$, $2^{15}-1$], 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 $\frac{1}{255}$, $\frac{1}{2^{16}-1}$, and $\frac{1}{2^{32}-1}$. 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 $instr_3$ (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 $instr_1$ the loop head, $instr_2$ the last instruction of the loop body and $instr_3$ 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 $\frac{1}{n}$, 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 $instr_1$ and $instr_2$ have a register define-use (def-use) relation, if $instr_1$ defines the value of a register (or some flag bit) and $instr_2$ 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 $\frac{1}{2}$ chance to write to some register or some flag bit (and the other $\frac{1}{2}$ chance writing only to 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/written-to according to the x86 instruction reference. As such, given two valid instructions $instr_1$ and $instr_2$, they have register def-use with the chance of $\frac{1}{2} \times \frac{1}{2^3} = \frac{1}{16}$. 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 a 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. 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, ... h_n converge on an instruction i, the probability of i representing data byte is $D[i] = p_1 \times p_2 \times ... \times p_n$. As such, when a large number of hints converge on i, i is highly unlikely a data byte.

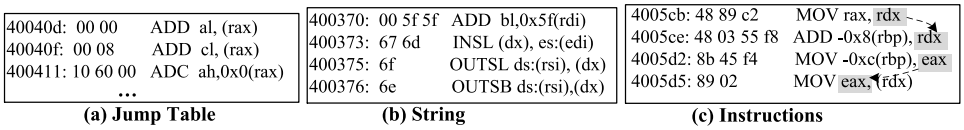

Fig. 5: *Register Definition-use Relation*

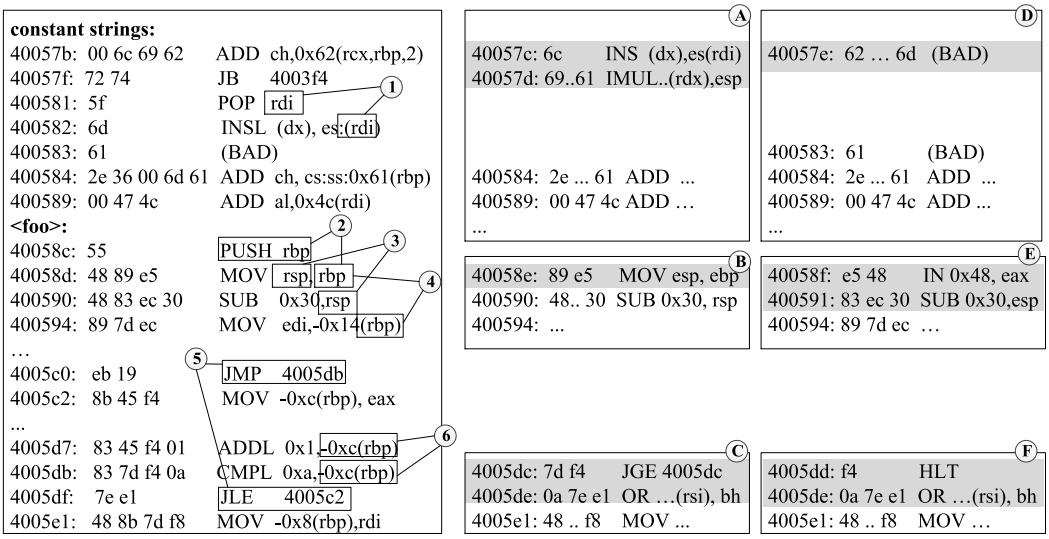

Fig. 6: Example for the algorithm; the code snippet in `foo()` corresponds to a statement "`for (i=0;i<11;i++) ...`"

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*[1].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, the key point is *that due to the occlusion rule, sequences quickly converge on true positives such that the occluded peers of the converged true positives are not reachable by any sequences and hence receive no hints.* As such, the true positives stand out in most cases, the exception being very short and featureless code segments. According to our experiments (see §V), our technique never misses any true positive and has as low as 3.7% false positives. In comparison, the false positive rate of superset disassembly is 85%.

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.

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 $fixed\_point$ 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 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

[1]This 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.

Algorithm 1 Probabilistic Disassembling

Input:	B - binary indexed by address H - probabilistic hints, denoted by a mapping from an address to a prior probability
Output:	$P[i]$ - posterior probability of an address i denoting a true positive instruction
Variable:	$D[i]$ - probability of address i being data byte $RH[i]$ - the set of hints, denoted by a set of addresses, that reach an address i

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: $RH[i] \leftarrow \{\}$
7: $fixed\_point \leftarrow false$
8: **while** $!fixed\_point$ **do**
9: $fixed\_point \leftarrow true$
▷ *Forward propagation of hints* (**Step I**)
10: **for** each address i from start of B to end **do**
11: **if** $D[i] \equiv 1.0$ **then**
12: continue
13: **if** $H[i] \neq \perp$ and $i \notin RH[i]$ **then**
14: $RH[i] \leftarrow RH[i] \cup \{i\}$
15: $D[i] \leftarrow \Pi_{h \in RH[i]} H[h]$
16: **for** each n, the next instruction of i along control flow **do**
17: **if** $RH[i] - RH[n] \neq \{\}$ **then**
18: $RH[n] \leftarrow RH[n] \cup RH[i]$
19: $D[n] \leftarrow \Pi_{h \in RH[n]} H[h]$
20: **if** $n < i$ **then**
21: $fixed\_point \leftarrow false$
▷ *Propagation to occlusion space* (**Step II**)
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: $D[i] \leftarrow 1\text{-}min_{j \text{ occludes with } i}(D[j])$
▷ *Backward propagation of invalidity* (**Step III**)
25: **for** each address i from end of B to start **do**
26: **for** each p, the preceding instruction of i along control flow **do**
27: **if** $D[p] \equiv \perp$ or $D[p] < D[i]$ **then**
28: $D[p] \leftarrow D[i]$
29: **if** $p > i$ **then**
30: $fixed\_point \leftarrow false$
▷ Compute posterior probabilities by normalization
31: **for** each address i from start of B to end **do**
32: **if** $D[i] \equiv 1.0$ **then**
33: $P[i] \leftarrow 0$
34: continue
35: $s \leftarrow \frac{1}{D[i]}$
36: **for** each address j, representing an instruction occluded with i **do**
37: $s \leftarrow s + \frac{1}{D[j]}$
38: $P[i] \leftarrow \frac{1/D[i]}{s}$

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 call targets*, 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 `0x4005c0` 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 `0x40057b` 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 Ⓐ-Ⓕ 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 corresponding instructions in the leftmost box. For example, in box Ⓐ, 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 `rdi`. 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 `rsp`, 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, $D[\texttt{0x400583}] = D[\texttt{0x40057e}] = 1.0$ 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 Ⓐ and Ⓓ 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, $D[0x40058d] = (\frac{1}{16})^3$ due to the three hints it is involved in. As shown in boxes Ⓑ, its occluded peer `0x40058e` cannot be reached from `0x40058c`. As a result, it gets no hint and $D[0x40058e] = \perp$. Similarly $D[0x40058f] = \perp$. Let us skip a few instructions and consider `0x4005db`. Due to the loop (with the backedge `0x4005df`→`0x4005c2`), hints ②-⑥ all reach `0x4005db`. As such, $D[0x4005db]$ is a tiny value smaller than $\frac{1}{2^{32}}$. In contrast, as shown in Ⓒ and Ⓕ, 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, $D[0x40058f] = D[0x40058e] = 1 - \frac{1}{16^3}$ and $D[0x4005dc] = D[0x4005dd] \simeq 1$.

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 $D[0x40058d] = \frac{1}{16^3}$, which is smaller than $D[0x40058b]$, there is no backward propagation. Although $D[0x40058e] = 1 - \frac{1}{16^3}$ 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 $D[0x40058b] = \frac{15}{16}$, $D[0x40058c] = \frac{1}{16}$. $P[0x40058c] = \frac{16}{16+16/15} = 0.94$ and $P[0x40058b] = 0.058$. The other true positive instructions have higher than 0.99 probabilities. For instance, $P[0x40058d] \simeq 0.9987$ and $P[0x40058e] = P[0x40058f] \simeq 0.0006$. $P[0x4005db] \simeq 1.0$ and $P[0x4005dc]$, $P[0x4005dd]$ 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 $P >= 0.01$, 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;

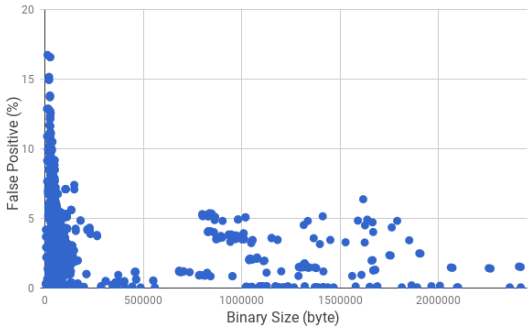

Fig. 7: Binary Size and FP rate

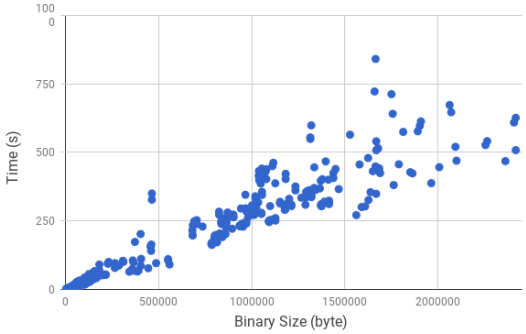

Fig. 8: Size and Processing Time

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 α (the x axis). For example, at the starting point on the left is $\alpha = 0.67\%$ (i.e., we keep instructions with $P >= 0.0067$), FP is about 4% and FN is 0, and 53.23% of the 607,758 functions in the corpora are

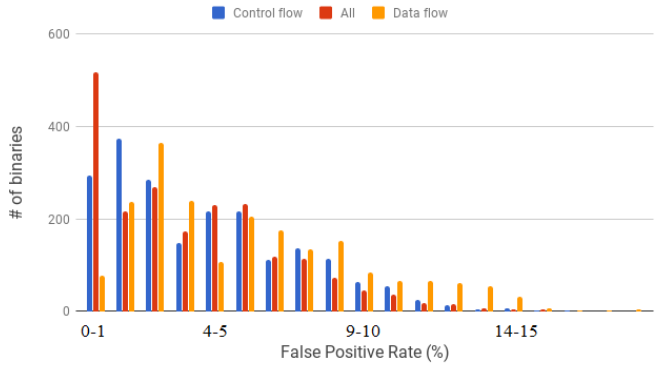

Fig. 9: Distribution of Different Kinds of Hints

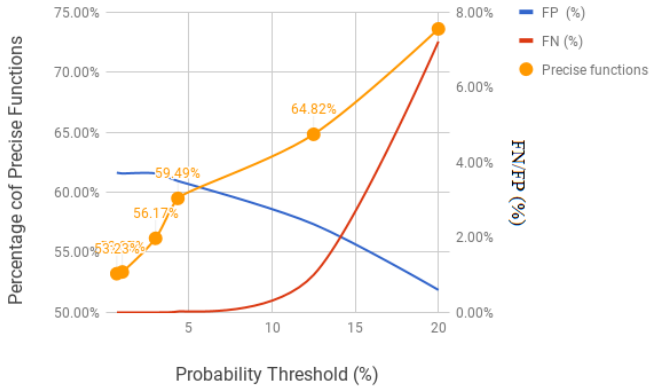

Fig. 10: Tradeoffs of Threshold Setting

precisely disassembled. With the growth of α, 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). However, ELF code generated by gcc on recent Ubuntus does not have as many such interleavings as on other platforms. Therefore, we simulate data-code interleavings by finding the unused space between functions in a number of binaries and replacing the unused

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

TABLE II: Superset Disassembly vs Probabilistic Disassembly

Binary Name	Objdump FN	Prob. Disassembly (PD) FP	PD FN
chroot	8	0.69%	0
xargs	10	2.51%	0
make-prime-list	10	3.49%	0
true	8	3.24%	0
false	10	3.34%	0
printenv	10	2.75%	0

TABLE III: Effectiveness with Interleaved Code and Data

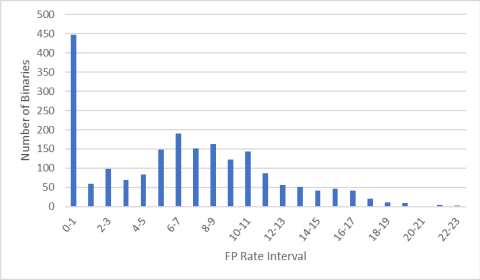

Fig. 11: FP Rates In the Absence of Function Entries

bytes (usually filled with no-ops) with random data bytes. Then we use both objdump (i.e., a linear sweep disassembler) and our disassembler to disassemble the transformed binaries. The results are presented in Table III. The second column presents the function entries missed by objdump due to occlusion caused by the mutation. In binary rewriting, missing entries is not acceptable. In contrast, our disassembler has 0 FN and very few false positives (columns 3 and 4).

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 I). 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 experiment for the comparison with superset disassembly, the benchmarks may not represent all features of real-world binaries. We will test our technique on more binaries. In our experiment of the code and data interleavings, we introduced the interleavings due to the lack of such cases in our benchmarks. We plan to port our technique to Windows, on which interleavings are very common. 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.

REFERENCES

[1] D. L. Bruening, *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[2] M. Smithson, K. Anand, A. Kotha, K. Elwazeer, N. Giles, and R. Barua, "Binary rewriting without relocation information," tech. rep., U. Maryland, 2010.

[3] M. P. T. cker Chiueh, *A Binary Rewriting Defense against Stack based Buffer Overflow Attacks*. SUNY Stony Brook, 2003.

[4] A. M., B. M., E. U., and L. J.;, *Control-flow integrity: principles, implementations, and applications*. University of California, Santa Cruz; Microsoft Research.

[5] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A.-R. Sadeghi, *MoCFI: A framework to mitigate control-flow attacks on smartphones*. University of California, Santa Barbara, 2012.

[6] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*, (Orlando, FL), December 2012.

[7] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries.," in *NDSS*, 2015.

[8] V. Van der Veen, D. Andriesse, E. Goktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrid, "Practical context-sensitive cfi," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, (Denver, Colorado), ACM, October 2015.

[9] V. van der Veen, E. Goktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, (San Jose, CA, USA), IEEE, May 2016.

[10] X. Chen, H. Bos, and C. Giuffrida, "CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks," in *EuroS&P*, Apr.

[11] L. Li and C. Wang, "Dynamic analysis and debugging of binary code for security applications," in *RV'13*.

[12] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA'09)*.

[13] F. Peng, Z. Deng, X. Zhang, and Z. S. Dongyan Xu, Zhiqiang Lin, "X-force: Force-executing binary programs for security applications," in *USENIX Security Symposium*, 2014.

[14] I. G. T. M. A. R. Guanhua Wang, Sudipta Chattopadhyay, "oo7: Low-overhead defense against spectre attacks via binary analysis," in *CoRR abs/1807.05843*, 2018.

[15] M. Bhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *ACM Conference on Computer and Communications Security (CCS'16)*.

[16] V.-T. Pham, M. Bhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *ASE'16*.

[17] V.-T. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury, "Hercules: Reproducing crashes in real-world application binaries," in *ICSE'15*.

[18] D. Gopan, E. Driscoll, D. Nguyen, D. Naydich, A. Loginov, and D. Melski, "Data-delineation in software binaries and its application to buffer-overrun discovery," in *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*.

[19] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: Security patch analysis for binaries towards understanding the pain and pills," in *ICSE'17*.

[20] E. Tilevich and Y. Smaragdakis, "Binary refactoring: Improving code behind the scenes," in *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*.

[21] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11)*.

[22] I. Gotovchits, D. Brumley, J. Bosamiya, and R. V. Tonder, "Binary analysis platform."

[23] Hex-Rays, *The Interactive Disassembler*.

[24] O. Yuschuk, "Ollydbg," *http://www. ollydbg. de/*, 2007.

[25] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *International Conference on Computer Aided Verification*, pp. 423–427, Springer, 2008.

[26] M. Smithson, K. Anand, A. Kotha, K. Elwazeer, N. Giles, and R. Barua, *Second Write: Binary Rewriting without Relocation Information*. University of Maryland, 2010.

[27] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pp. 9–16, ACM, 2011.

[28] X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 24–35, ACM, 2016.

[29] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, *BYTEWEIGHT: Learning to Recognize Functions in Binary Code*. Carnegie Mellon University; University of Chicago, 2014.

[30] E. Bauman, Z. Lin, and K. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*, (San Diego, CA), February 2018.

[31] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries.," in *USENIX Security Symposium*, pp. 583–600, 2016.

[32] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt, "Learning to analyze binary computer code.," in *AAAI*, pp. 798–804, 2008.

[33] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks.," in *USENIX Security Symposium*, pp. 611–626, 2015.

[34] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, pp. 177–189, IEEE, 2017.

[35] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in cots binaries," in *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, pp. 201–212, IEEE, 2017.

[36] R. Qiao, *Accurate Recovery of Functions in COTS Binaries*. PhD thesis, State University of New York at Stony Brook, 2017.

[37] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, 2003.

[38] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[39] P. Guide, "Intel® 64 and ia-32 architectures software developers manual," *Volume 3B: System programming Guide, Part*, vol. 2, 2011.

[40] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, (Seattle, WA, USA), pp. 189–200, ACM, 2008.

[41] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, (Dublin, Ireland), pp. 75–86, ACM, 2009.

[42] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu, "Dimsum: Discovering semantic data of interest from un-mappable with confidence," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, (San Diego, CA), February 2012.

[43] 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)* (D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis, eds.), vol. 3, (Athens, Greece), pp. 522–536, September 2011.

[44] R. Wartell, Y. Zhou, K. W. Hamlen, and M. Kantarcioglu, "Shingled graph disassembly: Finding the undecideable path," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 273–285, Springer, 2014.

[45] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pp. 265–275, IEEE, 2003.

[46] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," in *Proc. 37th IEEE/ACM International Sym. on Microarchitecture*, pp. 81–92, 2004.

[47] Nanda, W. Li, and L.-C. Lam, *BIRD: binary interpretation using runtime disassembly*. SUNY, Stony Brook, 2006.

[48] M. Smithson, K. ElWazeer, K. Anand, A. Kotha, and R. Barua, "Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 52–61, IEEE, 2013.

[49] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng, "Binary code continent: Finer-grained control flow integrity for stripped binaries," in *Proceedings of the 31st Annual Computer Security Applications Conference*, pp. 331–340, ACM, 2015.

[50] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, and R. B. A. D. Keromytis, *Retrofitting Security in COTS Software with Binary Rewriting*. University of Maryland; Columbia University, 2011.

[51] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, *Static Disassembly of Obfuscated Binaries*. Reliable Software Group; University of Santa Barbara, 2004.

[52] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *USENIX security Symposium*, vol. 13, pp. 18–18, 2004.

[53] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling.," in *USENIX Security Symposium*, pp. 627–642, 2015.

[54] S. Wang, P. Wang, and D. Wu, "Uroboros: Instrumenting stripped binaries with static reassembling," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1, pp. 236–247, IEEE, 2016.

[55] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," 2017.

[56] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis," in *European Symposium on Research in Computer Security*, pp. 481–500, Springer, 2008.

[57] G. Hoglund and J. Butler, *Rootkits: Subverting the Windows Kernel*, ch. 4: The Age-Old Art of Hooking, pp. 73–74. Pearson Education, Inc., 2006.