

Disassembly as Weighted Interval Scheduling with Learned Weights

Antonio Flores-Montoya*, Junghee Lim *, Adam Seitz *, Akshay Sood *, Edward Raff † and James Holt ‡

**GammaTech Inc.*

†*Booz Allen Hamilton*

‡*Laboratory for Physical Sciences*

Abstract—Disassembly is the first step of a variety of binary analysis and transformation techniques, such as reverse engineering, or binary rewriting. Recent disassembly approaches consist of three phases: an exploration phase, that overapproximates the binary’s code; an analysis phase, that assigns weights to candidate instructions or basic blocks; and a conflict resolution phase, that downselects the final set of instructions. We present a disassembly algorithm that generalizes this pattern for a wide range of architectures, namely x86, x64, arm32, and aarch64. Our algorithm presents a novel conflict resolution method that reduces disassembly to weighted interval scheduling. Additionally, we present a weight assignment algorithm that allows us to learn optimal weights for the various disassembly heuristics in the analysis phase. Learned weights outperform manually tuned weights in most cases while reducing the number of necessary heuristics by 40% (by setting their weights to zero). Our implementation, built on top of Ddisasm, outperforms state-of-the-art disassemblers in several metrics and achieves the largest proportion of perfectly disassembled binaries by a wide margin in all evaluated datasets.

1. Introduction

Disassembly is the process of recovering assembly instructions from binary code. This amounts to deciding, for each byte in the binary program, whether it belongs to an instruction or should be interpreted as data. Disassembly is an essential first step of a variety of binary analysis and transformation techniques, including reverse engineering, binary rewriting, or vulnerability discovery.

Traditional approaches rely on linear sweep or recursive disassembly. Linear sweep starts decoding instructions at the beginning of the code section and proceeds sequentially. This approach results in errors when data is interleaved with code. Recursive disassembly recovers the assembly code by traversing the control flow of the program, but is disadvantaged in resolving indirect branches limiting its coverage.

Modern reverse engineering frameworks [3], [6], [8], [13], [16], [22], [29] often use a combination of both linear and recursive disassembly, relying on heuristics to find additional code or make decisions among conflicting

code blocks [23]. These techniques and heuristics are often Instruction Set Architecture (ISA) specific. Several of the more recent techniques [15], [21], [26], [32] can be thought of as having the following steps:

- 1) An exploration phase in which a superset of all the potential instructions or data blocks is collected.
- 2) An analysis phase in which static analysis is performed to gather evidence for and against candidate instructions or data blocks, usually in the form of dependencies between candidates or weights assigned to each candidate.
- 3) A conflict resolution phase in which the evidence is aggregated to reach a final decision.

For example, D-ARM [32] proposes an approach for ARM in which the exploration phase considers all possibilities, i.e. every aligned address can be ARM code, Thumb code, or data; and expresses the conflict resolution problem as *maximum weight independent set* (MWIS) optimization problem [28]. Note that MWIS is NP-hard and D-ARM’s implementation needs to resort to a greedy approximation.

Ddisasm [15], D-ARM [32], and other comparable approaches suffer what we will refer to as the *weight assignment problem*. Given a set of analyses, heuristics and a conflict resolution method, what are the optimal weights that should be assigned to each of the heuristics to maximize performance? Are all heuristics necessary? Previous approaches rely on hand-tuned weights [15] or ISA-specific statistical studies combined and ad-hoc aggregation methods [26].

In our work, we present a multi-ISA (with support for x86, x64, arm32, and aarch64 binaries) disassembly approach that follows the three generic steps described above, but presents two key innovations. First, it relies on a tractable yet powerful conflict resolution algorithm. Our disassembly algorithm expresses the conflict resolution problem as a Weighted Interval Scheduling (WIS) problem, for which there is a complete and efficient dynamic programming algorithm [18]. In this setting, each code block or data block is considered a “task” to be scheduled with the restriction that tasks cannot overlap with each other. The resulting disassembly corresponds to the optimal schedule (the schedule with maximum aggregated weight).

Second, given this conflict resolution algorithm, we provide a solution to the weight assignment problem by encoding the inference of optimal heuristic weights with

respect to a set of binaries with known ground truth as a linear programming (LP) problem with soft constraints. This allows us to obtain optimal heuristic weights using off-the-self solvers (like Pulp [2]) even in cases when no perfect solution exists.

We build our implementation on top of Ddisasm [15] and evaluate it in comparison with other state-of-the-art tools on several ISAs. We perform this evaluation while accounting for the limitations of the existing approaches to gather ground truth [19], [24]. In addition, we evaluate individual aspects of the algorithm: the completeness of the exploration phase and the generalization properties of the weight inference algorithm. Finally, we discuss some insights about the learned heuristics weights.

In summary, our contributions are the following:

- 1) A multi-ISA disassembly algorithm that performs conflict resolution among candidate instructions and data blocks by solving a weighted interval scheduling (WIS) optimization problem (Sect. 3).
- 2) A weight assignment algorithm that optimizes the weights assigned to the heuristics used in the disassembly algorithm based on a corpus of ground-truth annotated binaries (Sect. 4). This algorithm supports incomplete annotations to account for the limitations in ground truth extraction techniques.
- 3) An extensive experimental evaluation of the disassembly algorithm and its corresponding weight assignment algorithm that includes a large collection of arm32, aarch64, x86, and x64 stripped binaries compiled with various compilers, optimization levels, and binary formats (ELF and PE) including the datasets from [17], [19], [20], [24] (Sect. 5).

2. Static Disassembly Instruction Recovery

In this section, we define static disassembly and state some simplifying assumptions that frame the problem.

2.1. Basic Definitions

Informally, we refer to disassembly as the process of recovering assembly instructions from binary code, which amounts to deciding the location and decode mode of the assembly instructions in the binary’s executable sections.

We restrict ourselves to *static* disassembly, i.e., we only consider instructions present in the binary file. In other words, we do not consider dynamically generated or self-modifying code.

Let us first define decode mode and instruction decoding. The default decode mode for any ISA will be referred to as A , and arm32’s second decode mode (Thumb, as opposed to ARM) will be referred to as T . Instruction decoding maps a sequence of bytes and a decode mode to an assembly instruction. As such, instruction decoding is deterministic, i.e. an address and a decode mode uniquely determine the instruction located at that address and its size. Thus, given a binary section spanning addresses $[s, e)$, let $d \in \{A, T\}$ be

a decode mode and $a \in [s, e)$ an address within the binary section. We define a partial function $instr(a, d)$ that defines the assembly instruction starting at address a and using the decode mode d^1 . The function $end(instr(a, d))$ denotes the end address of the instruction $instr(a, d)$.

Given this characterization of instructions, we define a binary’s *Code* as a set of instructions based on the binary program semantics.

Definition 1 (Code). An instruction $instr(a, d)$ is code $instr(a, d) \in Code$ if there exists a program trace that contains the instruction $instr(a, d)$.

Under this definition, unreachable instructions (dead code) are not considered code, and determining code accurately is not decidable (since it is connected to reachability). Engel et al. [14]’s sound disassembly decidability theorems allow overapproximating *Code* (which is indeed decidable in our setting), whereas our goal is to recover it precisely.

This definition also accounts for the challenges in collecting ground truth for disassembly [19], [24]. Developers may encode instructions meant to be executed as data, which confuses compiler-based ground truth generation.

Similarly, we can define data in a binary program based on its dynamic behavior.

Definition 2 (Data). A program address a is *data*, $a \in Data$, if there exists a program execution trace that reads or writes that memory location.

Instruction recovery aims to infer *Code*. In general, *Code* and *Data* are neither complementary nor mutually exclusive. There can be regions of a binary that are neither code (these regions are never executed) nor data (those bytes are never read or written). We call such regions *padding* since they are introduced by compilers to ensure alignment.

Example 1. The assembly listing in Figure 1 contains padding in the address range $[94c67 - 94c70)$. Those bytes are never executed² nor are they read by any instructions. Some disassemblers will interpret them as nops (as in the figure) or as data but they are neither according to our definition.

A binary region can also be both code and data. The instructions in that region are executed and also read or written during the lifetime of the program.

2.2. Assumptions

In the previous subsection, we have provided a general definition of *Code* and the desired output of the disassembly process. Next, we explicitly state two simplifying assumptions. In general, we find that while padding is extremely common in compiler-generated binaries, binary regions that

1. $instr(a, d)$ is partial because there might be combinations of decode modes and addresses that do not define a valid instruction in the given ISA.

2. Actually proving that those bytes are never executed would be very challenging but we are fairly confident in this particular case. The execution cannot fall through $94c67$ and we can find jumps that target $94c70$ directly.

```

94c40: je      94c43

94c42: lock  cmpxchg %rcx,0x13c6dd(%rip)
94c4b: cmp   %rdx,%rax
94c4e: je    94cfc
94c54: mov   0x13c6cd(%rip),%rdx
94c5b: mov   0x14380e(%rip),%rax
94c62: jmpq  94ad4

94c67: nopw  0x0(%rax,%rax,1)
94c70: mov   0x870(%r12),%r12

```

Figure 1. x64 assembly snippet extracted from `glibc-2.36`. The example contains both padding in the address range [94c67–94c70]) and a prefix-enclosed instruction at address 94c43.

are both code and data are uncommon. Thus, our disassembly algorithm rests on the following assumption:

Assumption 1. *Code and Data are mutually exclusive.*

Thus, our algorithm will approximate *Data* to better determine *Code*.

Our second assumption concerns overlapping instructions. Overlapping instructions are uncommon, but present in certain notable examples. In particular, we are aware of certain instances of overlapping instructions in `glibc` (e.g., see Figure 1). These overlapping instructions follow a very specific pattern in which an instruction can be executed with and without a prefix. We call these cases *prefix-enclosed instructions*.

Example 2. The assembly in Figure 1 contains a prefix-enclosed instruction at address 94c43. The conditional jump `je 94c43` will fall through to address 94c42 or jump to address 94c43 depending on the condition flag. This will result in the execution of `lock cmpxchg %rcx,0x13c6dd(%rip)` or `cmpxchg %rcx,0x13c6dd(%rip)`, respectively.

Assumption 2. *All instructions in Code are non-overlapping except for prefix-enclosed instructions.*

Note that most disassembly algorithms assume code and data are mutually exclusive and non-overlapping instructions [15], [21], [26], [32] though these assumptions are often not explicitly stated.

3. Disassembly Algorithm

3.1. Overview

Our disassembly algorithm relies on the concept of code blocks and data blocks. These are sequences of consecutive instructions or addresses containing data that are treated as a single entity.

Definition 3 (Code Block). A code block is a sequence of contiguous instructions with the same decode mode. Let $[s, e)$ be an address range and d be a decode mode, we can define a code block $codeBlock(s, e, d)$ as a sequence:

$$[instr(s_1, d), instr(s_2, d), \dots, instr(s_n, d)]$$

such that $s_1 = s$, $end(instr(s_i, d)) = s_{i+1}$ for all $1 \leq i < n$ and $end(instr(s_n, d)) = e$. Let b be a code block, $Insns(b)$ denotes the set of all the instructions in b . Conversely, $BlockOf(instr(s, d))$ defines the inverse mapping, i.e. the set of blocks that contain instruction $instr(s, d)$.

Definition 4 (Data Block). A data block $dataBlock(s, e)$, is a sequence of contiguous addresses in the range $[s, e)$.

We define a unified *block* notation to represent both code and data blocks. For that purpose we define a special “data” (denoted D) decode mode: i.e.,

$$\begin{aligned}
block(s, e, D) &= dataBlock(s, e) \\
block(s, e, d) &= codeBlock(s, e, d)
\end{aligned}$$

Given this definition, the result of the disassembly algorithm is a set of blocks $block([s, e), d) \in Blocks$ which can be mapped to *Code* and *Data* sets as defined in Sect. 2.1.

The disassembly algorithm has three phases:

- 1) **Candidate Generation:** The candidate generation phase generates a set of candidate blocks $CBlocks$ in which the code blocks should be an overapproximation of the final code blocks. It does so by performing a traversal of the binary that combines linear sweep and recursive disassembly.
- 2) **Block Weight Assignment:** The block weight assignment phase performs a series of analyses and assigns a weight to each of the candidate blocks based on heuristics. That is, it defines a function from candidate blocks to integers $W : CBlocks \rightarrow \mathbb{Z}$ where $W(block([s, e), d))$ is the weight associated to the candidate block $block(s, e, d)$.
- 3) **Conflict-Resolution:** The conflict resolution phase selects a subset of blocks $Blocks$ from $CBlocks$ ($Blocks \subseteq CBlocks$) based on their weight and such that there are no overlaps among the selected blocks.

Grouping instructions and data bytes into blocks serves two purposes. (1) It decreases the number of overall candidates for improved performance, and (2) it provides a coarser granularity for the analysis and conflict resolution phase. Heuristics can consider the likelihood of sequences of instructions rather than just individual instructions. This is particularly relevant for x86/x64, which has a dense instruction set and unaligned and variable-size instructions. For example, let us consider a `libc` x64 implementation. Its code section is 1.1MB and its superset disassembly contains 1,128,485 instructions with an average size of 3.2 bytes per instruction. If we generate candidates for each instruction and data byte, we will have approximately 2M candidates and 5M overlaps. In contrast, our candidate generation algorithm produces 100,820 candidates and only 5430 overlaps.

The following subsections describe each of the disassembly phases in detail.

3.2. Candidate Generation

The candidate generation phase generates a set of candidate blocks *CBlocks* and its goal is to ensure that the set is an overapproximation of the final blocks.

Our candidate generation algorithm extends Ddisasm’s original algorithm [15]. Ddisasm is implemented in Datalog, and so is our algorithm. Datalog can easily implement code traversals using recursive rules and pattern-matching rules for detecting code patterns (e.g. jump table patterns).

In Sect. 3.2.1 we summarize the original algorithm, and Sect. 3.2.2 describes how this algorithm has been extended. Sect. 3.2.3 formally defines the conditions that ensure that our candidate set *CBlocks* is overapproximating and characterizes possible failures during candidate generation.

3.2.1. Ddisasm’s original algorithm. The starting point of Ddisasm’s original disassembly algorithm is the superset of all instructions, i.e. it computes $instr(s, d)$ for every address s and decode mode d .

Ddisasm performs two traversals over this representation: a backward traversal conservatively discards instructions that lead (fallthrough or jump) to invalid locations, and a forward traversal generates candidate code blocks on the remaining instructions.

The forward traversal combines recursive and linear disassembly. The traversal of a binary program starts from an initial set of addresses, which is aggressively computed by considering the entry point of the program, symbols, exception information, and any sequence of bytes that could be interpreted as an address anywhere in the binary.

From those starting points, the traversal follows the control flow of the program. Whenever it encounters a jump or a call, it generates new starting points both at the jump/call target (if it is direct) and immediately after the jump. Starting points are added even after unconditional jumps or calls that are known not to return. That constitutes the “linear” component of the traversal. This traversal does not attempt to resolve indirect jumps or calls.

A post-processing phase then splits code blocks that have common suffixes, which ensures that code blocks have the following properties:

- 1) Candidate code blocks do not share instructions, i.e., each instruction belongs to at most one candidate block.
- 2) Each instruction within a code block must fall through into the next one in the block. This means that instructions that affect the control flow (e.g. jumps or function calls) or might stop the execution (e.g. `hlt`) can only be at the end of code blocks.

3.2.2. Extended Traversal. The original traversal was designed for Linux x64 binaries where data and code interleaving are rare [5]. Our extended traversal is designed to better support Windows x64, Linux arm32, and Linux aarch64 binaries. This extended support requires (1) the incorporation of multiple decode modes for arm32 (2) more

```
aeef8:  cmp r0, #211
aeefc:  bhi 0xd170
aef0:   adr r1, 0xaeef8
aef4:   ldr pc, [r1, r0, LSL #2]
aef8:   .word 0xcabc8
...
b16c:   .word 0xb248
```

Figure 2. Jump table snippet in arm32 binutils’s `strip` binary. The instruction at `0xaeef0` loads the jump table start address (`0xaeef8`), `r0` corresponds to the index variable, and the shift `LSL #2` implies that each jump table entry is 4 bytes. The comparison at address `aeef8` indicates that the jump table has 211 entries.

exhaustive handling of data and code interleavings. arm32 binaries make extensive use of literal pools [32], and Windows binaries (compiled with Visual Studio) often have jump tables in the code section.

For better handling of code and data interleavings, our extended forward traversal (1) generates candidate data blocks, and (2) extends its linear traversal.

Candidate Data block generation. The goal of data blocks is to “compete” against code blocks and help us discard spurious code blocks (relying on Assumption 1). Candidate data blocks are created whenever the traversal encounters code that accesses (reads or writes) memory locations in the code sections. We have specialized rules for creating candidate data blocks for the following situations:

- **Jump tables** We have multiple rules to detect different kinds of jump tables. Most jump tables follow a three-step pattern: (1) load the jump table start address, (2) use an index variable to compute the address of a jump table entry and load its content, and (3) perform an indirect jump to the computed address. Our detection rules focus on extracting the jump table start address, the size of the jump table entries, and the number of entries in the jump table.

Example 3. Figure 2 contains an example of a jump table in arm32 in which the starting point, entry size, and number of entries can be identified. Our algorithm will generate a data block candidate spanning from `0xaeef8` to $0xaeef8 + (212 * 4) = 0xb248$.

Unfortunately, it is sometimes challenging to determine a jump table size (the number of entries). If the jump table size cannot be determined, we start generating data block candidates at the jump table start address and with the size of the jump table entry, and continue generating candidates sequentially as long as: (1) The corresponding jump table entry points³ to potentially valid code. (2) The jump table entry does not overlap with any of the already traversed jump table targets or with another jump table start.

3. Note that a jump table entry does not necessarily contain an absolute address. It might contain a relative address in which case the target needs to be computed accordingly.

Address	Data	ARM	Thumb	Candidate blocks	Sorted by end address
19754:	bcd6	strheq sp, [r2], -ip		D A	2 1
19756:	0200		movs r2, r0		
19758:	1340	push {r0, r1, r4, lr}	ands r3, r2	T	3 4
1975a:	2de9		push.w {r5, r6, lr}		
1975c:	6040	ldr r4, [pc, #96]		A T	7 5
1975e:	9fe5		b #0x192a0		
19760:	0400	str r0, [sp, #4]	movs r4, r0	T	6
19762:	8de5		b #0x19280	T	8

Figure 3. arm32 Binary snippet extracted from program `procd` from the openwrt dataset [17]. The example illustrates different possible interpretations as Data (D), ARM (A), or Thumb (T) code of the address range [19754, 19764]. The right hand side contains a representation of all the candidate blocks generated by the extended traversal, annotated with the decode mode (left) and numbered according to their end address (right). The real blocks are gray.

This approach might overestimate the length of jump tables and result in spurious data block candidates which can be resolved by the conflict resolution phase. In practice, we believe this happens rarely since jump table targets are often located right after the jump table.

Example 4. Let us consider the jump table in Figure 2 again. If the jump table size was not inferred from the comparison at address `0xae8`, our algorithm will generate data block candidates of size 4 starting at `0xae8` and at 4 byte increments. At address `0xcbc8`, we know the target of the jump table entry `0xcbc8` points to code, and thus must be an upper bound on the extent of the jump table. As we traverse subsequent jump table entries, we tighten that upper bound with the discovered jump table targets. Once we reach the entry at address `b16c`, we update the jump table limit to `0xb248` which is indeed the end of the jump table.

- **Potential strings** Whenever we detect a memory access to a location that could contain a string (a sequence of valid ASCII characters that ends with `\0`), we create a data block candidate with the size of the potential string. For potential strings above a threshold (>8 bytes), we generate data block candidates even if no reference from the code is detected.
- **Repeated bytes** Whenever we find a sequence of repeated bytes over a certain length (>8 bytes), we create a data block candidate encompassing all the repeated bytes.
- **Other data accesses** Data accesses from the code that do not correspond to jump tables or strings fall in this category. In such cases, we create a candidate data block using the size of the data access. Note that it is often necessary to consider multiple instructions to determine memory accesses, especially for RISC architectures like arm32.

Extended linear traversal. The linear traversal continues traversing code after the end of code blocks. Our extension accounts for arm32 decode modes and data blocks.

The linear traversal in arm32 maintains the decode mode, i.e. it will attempt to decode instructions using the same decode mode (A or T) as the last visited code block.

However, if the traversal encounters invalid instructions (as determined by the backward traversal), it will try switching decode mode (e.g., from ARM to Thumb or vice-versa).

A linear traversal can cross padding instructions but might be interrupted by data interleavings (since data interleaving might not be interpretable as instructions). Thus, we initiate linear traversal *after all* candidate data blocks as well. For arm32, traversal after candidate data blocks is attempted with both ARM and Thumb decode modes.

Example 5. The example in Figure 3 contains a candidate data block `dataBlock(19754, 19758)`. This data block will trigger extended code traversals starting at address 19758 in both ARM and Thumb mode, resulting in candidate blocks 7, 4, 5, 6, and 8, respectively (see numbering on the right-hand side of Figure 3).

3.2.3. Sound Overapproximation. In this section, we formally define the conditions that make a candidate block set a sound overapproximation, discuss its failure modes, and how our candidate generation phase can result in errors.

Definition 5 (Sound Candidate Overapproximation). A candidate block set $CBlocks$ is a *sound overapproximation* of the code if there exists a subset $Blocks \subset CBlocks$ such that $\bigcup_{b \in Blocks} Insns(b) = Code$.

There are two situations that can result in a $CBlocks$ set that is not a sound overapproximation.

Missed instructions. There is some $instr(s, d) \in Code$ that does not belong to any candidate code block. This can happen if the forward code traversal does not visit those instructions. Missed instructions will inevitably lead to false negatives, i.e. real instructions that are not considered as code.

Incorrect code block boundaries. There is a candidate code block b in which some instructions are code and others are not, i.e. $Insns(b) \cap Code \neq \emptyset$ and $Insns(b) \not\subseteq Code$. This situation can happen if a sequence of spurious instructions falls through a sequence of real instructions, and no block limit can be inferred at the boundary. Incorrect code block boundaries will lead to false positives if the candidate block with incorrect boundaries is selected during conflict resolution or false negatives otherwise.

Example 6. Consider the example in Figure 3. If instruction $instr(19754, T)$ was code, it would be a missed instruction since it does not belong to any candidate code block.

If no traversal started at address 19758 (as described in Example 5), ARM instruction at 19754 would fall through to 19758, generating a single candidate block $codeBlock(19754, 19764, A)$ instead of the two separate candidates 2 and 7. Such a block would have incorrect boundaries since it would contain both real (at [19758 – 19764]) and spurious instructions (at 19754).

Theorem 1. *If a candidate block set $CBlocks$ does not present missed instructions nor incorrect code block boundaries, then $CBlocks$ is a sound overapproximation.*

Proofs for all theorems can be found in Appendix A.

The traversal outlined in this section does not guarantee that we generate a sound overapproximation. However, this characterization allows us to distinguish disassembly failures caused by the candidate generation phase as opposed to failures due to incomplete heuristics or inadequate heuristic weights. We measure the prevalence of these errors (missed instruction and incorrect block boundaries) experimentally in Sect. 5.1.

3.3. Block Weight Assignment

The goal of this phase is to compute the weight associated with each candidate block ($W(b)$ for each $b \in CBlocks$).

Our analysis defines a set of heuristic rules $h_j \in H$ that can match individual candidate blocks one or several times $\# : CBlocks \times H \rightarrow \mathbb{N}$ (where \mathbb{N} is the set of natural numbers). Each heuristic h_j has a corresponding integer weight $w_j \in \mathbb{Z}$. We distinguish two kinds of rules depending on how they contribute to a block’s weight: simple and proportional. Simple heuristics (H_s) contribute weight based solely on the number of matches and the heuristic weight, whereas proportional heuristics (H_p) contribute proportionally to the size of the candidate block in bytes.

Let $b \in CBlocks$, its overall weight is computed as follows.

$$W(b) = \sum_{h_j \in H_s} \#(b, h_j)w_j + \sum_{h_k \in H_p} \#(b, h_k)size(b)w_k \quad (1)$$

We extended Ddisasm to have 34 multi-ISA heuristics and 58 arm32-specific heuristics. While enumerating all the heuristics falls beyond the scope of this paper, heuristics can be grouped into the following categories:

- *Control flow:* Rules that assign points based on how a candidate code block is referenced or references other candidate code blocks.
- *Instruction patterns:* Rules that assign points based on likely or unlikely instruction patterns.
- *Metadata:* Rules that use existing metadata (when-ever available) such as symbols, relocations, or exception information. For example, instructions

overlapping with function symbols receive negative points.

- *Data references:* Rules that assign points to candidate data blocks that are referenced by candidate code blocks. This includes rules that correspond to jump table detection and literal pool accesses.
- *Data content:* Rules that assign points to candidate data blocks based on their content.
- *Surrounding context:* Rules that assign points to candidate blocks based on the surrounding candidate blocks. E.g. a code block that fits in between two other candidate code blocks is more likely to be real, or a literal pool entry is often surrounded by other literal pool entries.

3.4. Conflict Resolution

The conflict resolution phase selects a subset of blocks from $Blocks \subseteq CBlocks$ that maximizes the overall weight and such that there are no overlaps among the selected blocks (relying on the assumptions described in Sect. 2.2). Prefix-enclosed instructions (Example 2) are handled as a special case⁴. For the remainder of the paper, we assume that selected blocks $Blocks$ do not overlap with each other.

Maximizing the weights of the selected blocks while avoiding overlaps is a problem that can be encoded directly as a weighted interval scheduling problem. In previous sections, we have described how to compute a set of candidate blocks $b \in CBlocks$ and assign weights to each candidate block $W(b)$. These correspond to the input of the algorithm.

Each candidate block $b := block([s, e], d)$ corresponds to a task that needs to be “scheduled” which starts at address $start(b) = s$ and ends at address $end(b) = e$. The output of the algorithm is a set of selected blocks $Blocks$ that form the optimal schedule.

Algorithm 1’s implementation closely follows the textbook dynamic programming implementation of a weighted scheduling algorithm (e.g. [18]). The algorithm contains three high-level steps. First, we sort candidate blocks by increasing *end address*. If we have several candidate blocks with the same *end address* we consider (1) the start address and (2) the block mode ($A < T < D$) lexicographically to ensure a total order. Figure 3 presents an example of this ordering on its right-hand side. Given this ordering, we can compute the *predecessor* index $PRED(i)$ for each block b_i . b_i ’s predecessor index refers to the last block that precedes b_i and does not overlap with it (given the ordering of blocks by end address, we know that all preceding blocks do not overlap either).

Example 7. $PRED(i)$ has the following values for the candidate blocks in Figure 3:

⁴ Prefix-enclosed instructions result in two overlapping candidate blocks in which their first instruction has a prefix in one and no prefix in the other. We consider only one of the blocks (the one with a prefix) for conflict resolution. Whichever decision is taken for that candidate block (whether it is included in the final $Blocks$) is also adopted for the other candidate in a post-processing phase.

Algorithm 1 Block Conflict Resolution

```
1: function INTERVALSCHEDULING(CBlocks)
2:    $b_1, b_2, \dots, b_n = \text{SORTBYEND}(CBlocks)$ 
3:   function PRED(i)
4:     return  $\max(\{0\} \cup \{j \mid \text{end}(b_j) \leq \text{start}(b_i)\})$ 
5:   end function
6:   // Compute maximum weight
7:    $opt[0] = 0$ 
8:   for  $i = 1$  to  $n$  do
9:      $opt[i] = \max(W(b_i) + opt[\text{PRED}(i)], opt[i - 1])$ 
10:    if  $W(b_i) + opt[\text{PRED}(i)] \geq opt[i - 1]$  then
11:       $mem[i] = \text{PRED}(i)$ 
12:    else
13:       $mem[i] = i - 1$ 
14:    end if
15:  end for
16:  // Recover optimal schedule
17:   $Blocks = \emptyset$ 
18:   $i = n$ 
19:  while  $i > 0$  do
20:    if  $mem[i] == \text{PRED}(i)$  and  $W(b_i) \geq 0$  then
21:       $Blocks = Blocks \cup \{b_i\}$ 
22:       $i = mem[i]$ 
23:    else
24:       $i = i - 1$ 
25:    end if
26:  end while
27:  return  $Blocks$ 
28: end function
```

PRED(1) = 0 PRED(2) = 0 PRED(3) = 0 PRED(4) = 3
PRED(5) = 4 PRED(6) = 5 PRED(7) = 3 PRED(8) = 6

Second, the dynamic programming algorithm computes the maximum weight for all the scheduling subproblems up to block b_i in $opt[i]$. Each iteration considers whether a candidate block is included in the schedule or not. At each step, $mem[i]$ records that choice⁵. For example, in the interval scheduling problem from Figure 3, we will have $W(b_7) + opt[3] > opt[6]$, which results in b_7 being chosen over the optimal schedule up to block 6. Thus, we have $opt[7] = W(b_7) + opt[3]$ and $mem[7] = 3$. Finally, in the third step, we recover the optimal schedule (the final block set) by traversing $mem[i]$ backwards from the last candidate block. When $\text{PRED}(i) = i - 1$, the choice is between including b_i in the schedule or not, which will depend on whether b_i 's weight is positive or negative.

4. Heuristic Weight Assignment

Our disassembly algorithm (Sect. 3) assumes each heuristic $h_j \in H$ has a corresponding integer weight $w_j \in \mathbb{Z}$. In its initial implementation those weights were hand-picked based on the programmer's intuition and on manual evaluations. Unfortunately, this approach does not scale, and weight assignment becomes harder as new heuristics are

5. In case of a tie, Algorithm 1 selects the latter block (see \geq in Line 10).

developed or updated. As mentioned in the introduction, this problem is not exclusive to Ddisasm, but common to approaches that rely on weighted heuristics such as D-ARM [32].

In this section, we describe our automated approach for assigning optimal weights to the disassembly heuristics to maximize the disassembly's accuracy. This approach relies on having a collection of binaries annotated with ground truth information. In particular, it assumes the existence of *partial* ground truth information consisting of a set of instructions *TCode* and a set of addresses *Ignored* for which ground truth is not available. We discuss how ground truth is extracted for each dataset in Sect. 5, and the specific limitations that result in *Ignored* addresses in Appendix B.

4.1. Overview

Given a binary with ground truth, we can partially run the disassembly algorithm. This allows us to collect candidate blocks *CBlocks* and heuristic matches $\#(b, h)$ for each candidate block $b \in CBlocks$ and each heuristic $h \in H$.

In addition, we can map our ground truth information *TCode* and *Ignored* to two subsets of candidate blocks: *TBlocks* $\subset CBlocks$ and *FBlocks* $\subset CBlocks$ ("True" and "False" blocks respectively). We define *TBlocks* as:

$$TBlocks = \bigcup_{i \in TCode} BlockOf(i)$$

Let $start(Insns(b))$ denote the set of all starting addresses of instructions in a code block b . We then define *FBlocks* as follows:

$$FBlocks = \{b \mid b \in CBlocks \setminus TBlocks \\ \wedge start(Insns(b)) \not\subseteq Ignored\}$$

If *CBlocks* is a sound overapproximation (Definition 5), *TBlocks* contains all and only true instructions (*TCode*). *FBlocks* are candidate code blocks that are neither true blocks nor completely ignored. Note that there are candidate blocks that are neither *TBlocks* nor *FBlocks*. Those are blocks for which we do not have definitive ground truth. In particular, this applies to all the candidate data blocks⁶.

Example 8. In our example in Figure 3, $Code = \{instr(19758, A), instr(1975c, A), instr(19760, A)\}$ and $Ignored = \emptyset$, which results in $TBlocks = \{b_7\}$ and $FBlocks = \{b_1, b_{3-6}, b_8\}$. Candidate data block b_2 does not belong to either.

Rather than performing interval scheduling (conflict resolution) to find the block selection based on a given set of fixed weights, we need to find out which weights will lead to the right decisions during interval scheduling. The correct

6. As opposed to candidate code blocks, which are guaranteed not to share instructions (Sect. 3.2.1), we do not enforce any properties on candidate data blocks. For a region of the binary that should be considered data, there might be multiple competing data blocks and our ground truth does not tell us which ones to pick.

decisions involve selecting all candidate blocks in $TBlocks$ and not selecting any blocks in $FBlocks$. Schedules that satisfy those conditions are *optimal*.

Definition 6 (Optimal schedule). A candidate block set S is an *optimal* schedule if $TBlocks \subseteq S$ and $FBlocks \cap S = \emptyset$.

We propose a new algorithm (Sect. 4.2) that performs weighted interval scheduling symbolically (with symbolic weights) and uses the ground truth information to collect a set of linear constraints that enforce the selection of an optimal schedule. For clarity, we first present a naive implementation for inferring constraints from a weighted interval scheduling problem in Sect. 4.2 and later present an optimized implementation in Sect. 4.3.

We apply this algorithm to a collection of binaries and infer a constraint set for each binary. The overall constraint set is not guaranteed to be satisfiable—It is possible that there is no weight assignment that satisfies all the constraints, which is an indication that additional heuristics might be needed or some heuristic rules might need to be refined. Thus, we encode the problem as a linear programming (LP) optimization with soft constraints to find a weight assignment that maximizes the number of satisfied constraints (see Sect. 4.4).

4.2. Weight Constraints Inference

The naive algorithm for inferring constraints (Algorithm 2) follows a similar structure as Algorithm 1. They both iterate over all candidate blocks sorted by their end address. While Algorithm 1 collects an optimal numerical value in each iteration $opt[i]$, the constraint inference algorithm collects a set of symbolic schedules in $symopt[i]$.

Let b be a candidate block, its symbolic weight is defined as $SW(b)$. We compute it with the same formula as $W(b)$ (Equation 1) but using symbolic values for the heuristic weights w_j . Instead of an integer, $SW(b)$ returns a linear expression of the form $c_1w_1 + c_2w_2 + \dots + c_mw_m$ where $c_j \in \mathbb{N}$ and w_1, w_2, \dots, w_m are variables representing the unknown heuristic weights.

Let S be a set of blocks representing a concrete schedule, we represent in our algorithm with a tuple $\langle SW(S), score(S) \rangle$. The first term $SW(S)$ represents the symbolic weight of the schedule, defined as $SW(S) = \sum_{b \in S} SW(b)$. The second term $score(S)$ is a numerical score based on the number of $TBlocks$ and $FBlocks$ that are included in the schedule.

$$score(S) = \begin{cases} -1 & \text{if } FBlocks \cap S \neq \emptyset \\ |TBlocks \cap S| & \text{otherwise} \end{cases} \quad (2)$$

This score captures our priority in selecting schedules. Optimal schedules are guaranteed to have maximal score.

Lemma 1. *Let S be a schedule, if S is optimal (Definition 6), its score is $score(S) = |TBlocks|$, otherwise $score(S) < |TBlocks|$.*

Lemma 1 follows directly from the definition of $score$ (Equation 2).

Algorithm 2 Naive Constraint Inference

```

1: function INFERCS( $CBlocks, TBlocks, FBlocks$ )
2:    $symopt[0] = \{0, 0\}$ 
3:   for  $i = 1$  to  $n$  do
4:      $take = \{ \langle SW(b_i) + s.weight, uScore(s.score, b_i) \rangle$ 
5:        $\mid s \in symopt[PRED(i)] \}$ 
6:      $leave = symopt[i - 1]$ 
7:      $symopt[i] = take \cup leave$ 
8:   end for
9:    $l = bestSched(symopt[n])$ 
10:   $rs = \{ r \mid r \in symopt[n] \wedge r.score < l.score \}$ 
11:  return  $\{ l.weight > r.weight \mid r \in rs \}$ 
12: end function

```

Algorithm 2's main loop amounts to the incremental computation of all the symbolic schedules. Let s be a symbolic schedule, we use $s.weight$ and $s.score$ to refer to the weight and score of s respectively. On each iteration i , we have a decision point: whether b_i is included in the schedule. We compute the two alternatives $take$ (when b_i is included) and $leave$ (when b_i is not included). To compute $take$ we need to add a symbolic expression corresponding to block b_i 's weight to all the schedules in $symopt[PRED(i)]$ (we reuse PRED's definition from Algorithm 1).

The score of the schedules in $take$ is updated using $uScore$ which corresponds to the incremental computation of $score$ (see Equation 2). Let s be a score and b a block under consideration, $uScore$ is defined as follows:

$$uScore(s, b) = \begin{cases} -1 & \text{if } s = -1 \vee b \in FBlocks \\ s + 1 & \text{if } b \in TBlocks \\ s & \text{otherwise} \end{cases} \quad (3)$$

The value of $symopt[i]$ corresponds to the maximum of all the potential schedules in $take$ and $leave$. Once we have computed $symopt$, $symopt[n]$ contains a symbolic representation of all the possible schedules (block selections) for the binary. In particular, it contains at least one optimal schedule. Given that our ground truth is partial and there can be overlapping candidate data blocks, there can be several optimal schedules. Function $bestSched$ returns an optimal schedule by leveraging Lemma 1. If there are several, it chooses one of them heuristically by selecting the optimal schedule with the highest sum of positive heuristic coefficients. Then, we generate linear constraints that ensure that our selected optimal schedule has a higher overall weight than every other non-optimal schedule (which are guaranteed to have a lower score by Lemma 1).

Example 9. Continuing from Example 8, let us assume we have 4 heuristic rules s (size), j (jumped), l (literal pool), and c (called) with unknown weights w_s, w_j, w_l , and w_c respectively. Assume further that s is a proportional rule and all the others are simple, each block matches $\#(b, s) = 1$, and we have $\#(b_2, l) = 1$, $\#(b_3, j) = 1$, $\#(b_5, j) = 1$, and

$\#(b_7, c) = 1$. All other combinations of blocks and rules yield zero. In this scenario, we have the following symbolic weights for each of the blocks:

$$\begin{aligned} SW(b_1) &= 4w_s & SW(b_5) &= 6w_s + w_j \\ SW(b_2) &= 4w_s + w_l & SW(b_6) &= 2w_s \\ SW(b_3) &= 2w_s + w_j & SW(b_7) &= 12w_s + w_c \\ SW(b_4) &= 2w_s & SW(b_8) &= 2w_s \end{aligned}$$

At the end of the loop, *symopt*[8] contains symbolic schedules representing all the combinations of blocks. In particular, it contains a tuple representing the optimal schedule $\{b_2, b_7\}$: $\langle 16w_s + w_l + w_c, 1 \rangle$, as well as non-optimal schedules, such as $\{b_3, b_7\}$: $\langle 14w_s + w_j, -1 \rangle$, which should not be selected. Based on those two schedules, Algorithm 2 generates a linear constraint $16w_s + w_l + w_c > 14w_s + w_j$. Note that even in this example, there are two optimal schedules $\{b_7\}$ and $\{b_2, b_7\}$.

Theorem 2. *Let C_s be the set of constraints generated by Algorithm 2 and let $\alpha : H \rightarrow \mathbb{Z}$ be a weight assignment that satisfies the constraints $\alpha \models C_s$, then there is an optimal schedule S such that $\alpha(SW(S)) > \alpha(SW(S'))$ for every non-optimal schedule S' .*

Collorary 1. *Let C_s be the set of constraints generated by Algorithm 2 and let $\alpha : H \rightarrow \mathbb{Z}$ be a weight assignment that satisfies the constraints $\alpha \models C_s$, then the schedule selected by Algorithm 1 using weight assignment α is optimal.*

This theorem and corresponding corollary ensure that a satisfying weight assignment will lead to an optimal schedule—an optimal block selection with respect to the ground truth—being selected during conflict resolution.

The fact that we choose only one optimal schedule in *bestSched* and that optimal schedules are not necessarily unique (see Example 9) means that our algorithm is not complete. Even if we fail to find a satisfying weight assignment that selects an optimal schedule, there might exist one for a *different* optimal schedule. This is acceptable for our use case since our goal is to learn heuristic weights that work well in practice and minimize the number of errors even in situations where errors cannot be completely eliminated.

4.3. Optimized Constraint Inference

The number of potential schedules in *symopt*[i] can, in the worst case, double in each iteration. We alleviate this exponential growth risk by incorporating several optimizations. Algorithm 3 contains the most relevant optimizations.

4.3.1. Interval Scheduling Decomposition. Up to this point, we have considered the block conflict resolution as a single WIS problem that encompasses all candidate blocks in a binary. However, WIS problems can often be decomposed into smaller subproblems.

Example 10. Consider the interval scheduling problem in Figure 3. There are many possible schedules, e.g. $\{1, 7\}$, $\{2, 7\}$, $\{1, 4\}$, or $\{2, 4\}$. However, this problem can be decomposed into two completely independent decisions: (i) which intervals are chosen between from 1-3 (ii) which ones

are chosen from 4-8. None of the choices for blocks in 4-8 preclude us from choosing any of the blocks from 1-3. This means that we split this scheduling problem into two and generate constraints for each of the subproblems.

More formally, we can split an interval scheduling problem b_1, b_2, \dots, b_n on b_i if all $\text{PRED}(j) \geq i$ for $i < j \leq n$. In that case, we can recursively unfold *symopt*[n]'s definition until every term contains *symopt*[i], which can be factored out. Thus, we can compute the optimal schedule for b_1, b_2, \dots, b_i and for b_{i+1}, b_2, \dots, b_n separately and then combine them. In practice, an interval scheduling problem for a binary can often be split into many small independent subproblems. This splitting can lead to important savings in Algorithm 3. The size of *symopt* can grow exponentially (in the worst case) with the number of intervals considered (2^n), so reducing the size of n has a very significant effect. For example, the *procd* binary selected in Figure 3 has 4926 candidate blocks, but after decomposition, the largest scheduling subproblem contains only 11 candidates.

4.3.2. Schedule Subsumption. Despite not knowing the optimal values of the heuristic weights beforehand, it is reasonable to fix their sign. We can divide heuristics into positive and negative heuristics $H = H^+ \cup H^-$ ($w_j \geq 0$ if $h_j \in H^+$ and $w_j \leq 0$ if $h_j \in H^-$). This distinction allows us to perform further optimizations.

Let $e = c_1w_1 + c_2w_2 + \dots + c_mw_m$ be a linear expression where $c_j \in \mathbb{Z}$ and w_j are variables representing weights. We know that e is positive ($e \geq 0$) if $c_j \geq 0$ for all positive heuristics $h_j \in H^+$ and $c_j \leq 0$ for all negative heuristics $h_j \in H^-$. Then, given two partial schedules A and B , we say A *subsumes* B if we can prove $SW(A) \geq SW(B)$ (we know A 's weight will be higher than or equal to B 's regardless of the chosen weights) and $\text{score}(A) \leq \text{score}(B)$. The second condition $\text{score}(A) \leq \text{score}(B)$ means there are three possibilities:

- 1) Both A and B will be part of optimal schedules. In that case A is a better schedule for generating constraints. Any satisfying weight assignment α for B , will also be satisfying for A , but there might be additional possibilities for A .
- 2) Both A and B will be part of non-optimal schedules. Then, a constraint that ensures the weight of A is smaller than that of an optimal schedule, will also guarantee that B 's weight is smaller.
- 3) A is part of a non-optimal schedule and B part of an optimal schedule. In this case, any constraints trying to enforce $SW(B) > SW(A)$ will be trivially unsatisfiable.

In all the cases above, we can discard B (we do not need to take it into account for generating constraints). At each iteration, function *simplifyS* removes subsumed schedules from *symopt*[i] (Line 14).

Example 11. In the previous example, we decomposed the scheduling from Figure 3 into two, one for candidates 1 – 3 and another for candidates 4 – 8. Let us focus now on the

latter and assume the same heuristics and heuristic matches as Example 8. We also assume all heuristics are positive.

Without subsumption, $symopt[4]$ contains tuples $\langle 0, 0 \rangle$, $\langle 2w_s, -1 \rangle$ corresponding to schedules $\{\}$ and $\{4\}$ respectively. Since w_s is positive, we know that $2w_s \geq 0$ and $\langle 2w_s, -1 \rangle$ subsumes $\langle 0, 0 \rangle$ which can be discarded. After simplifying, $symopt[4]$ only contains the tuple $\langle 2w_s, -1 \rangle$.

A similar situation happens for $symopt[5]$. Without subsumption, $symopt[5]$ contains 4 tuples corresponding to schedules $\{\}$, $\{4\}$, $\{5\}$, and $\{4, 5\}$. However, they are all subsumed by the tuple corresponding to schedule $\{4, 5\}$: $\langle 8w_s + w_j, -1 \rangle$. Thus, the optimized version contains a single tuple.

Algorithm 3 Optimized Constraint Inference

```

1: function INFERCS( $CBlocks, TBlocks, FBlocks$ )
2:    $symopt[0] = \{(0, 0)\}$ 
3:    $Cs[0] = \emptyset$ 
4:   for  $i = 1$  to  $n$  do
5:      $take = \{(SW(b_i) + s.weight, uScore(s.score, b_i))$ 
6:        $\mid s \in symopt[PRED(i)]\}$ 
7:      $leave = symopt[i - 1]$ 
8:     if  $b_i \in TBlocks$  then
9:        $l = bestSched(take)$ 
10:       $newCs = \{l.weight > r.weight \mid r \in leave\}$ 
11:       $symopt[i] = \{l\}$ 
12:       $Cs[i] = simplifyC(Cs[PRED(i)] \cup newCs)$ 
13:     else
14:        $symopt[i] = simplifyS(take \cup leave)$ 
15:        $Cs[i] = simplifyC(Cs[PRED(i)] \cup Cs[i - 1])$ 
16:     end if
17:   end for
18:    $l = bestSched(symopt[n])$ 
19:    $rs = \{r \in symopt[n] \wedge r.score < l.score\}$ 
20:    $newCs = \{l.weight > r.weight \mid r \in rs\}$ 
21:   return  $simplifyC(newCs \cup Cs[n])$ 
22: end function

```

4.3.3. Constraint Subsumption. Similarly to schedule subsumption, we can detect constraint subsumption. Let $c_1 := l_1 > r_1$ and $c_2 := l_2 > r_2$ be two linear constraints. If $(l_2 - r_2) - (l_1 - r_1) \geq 0$, then $c_1 \Rightarrow c_2$ and c_2 can be discarded. This can be checked similarly to schedule subsumption by checking the coefficients of positive and negative heuristics in the linear expression $(l_2 - r_2) - (l_1 - r_1)$.

Function $simplifyC$ (Lines 12, 15 and 21) simplifies the generated constraint sets by removing subsumed constraints.

4.3.4. Partial Schedule Constraints. Whenever we encounter a $b_i \in TBlocks$ in the main loop, we know that b_i must be part of all optimal schedules, and the decision needs to happen based on the currently accumulated weight. Thus, we can generate constraints based on those partial schedules and discard the partial schedules that do not select b_i and thus are guaranteed to be non-optimal (see Lines 9-11 in Algorithm 3). These constraints are accumulated in Cs

throughout the loop, and we add them to the final constraint set at the end. This optimization reduces the growth of $symopt$, given that on each iteration that corresponds to a true block $b_i \in TBlocks$, we reset the size of $symopt[i]$ to 1. It can also result in a smaller number of overall constraints.

4.4. Solving Weight Constraints

We can repeat the procedure described in the previous section (Sect. 4.2) to obtain a constraint set for each binary with ground truth. We accumulate those constraints for a collection of binaries and try to find a weight assignment that satisfies all constraints. However, this might not be feasible. If an optimal weight assignment does not exist, this is an indication that the existing heuristics are insufficient to handle all possible cases. Nevertheless, we want to find an assignment that minimizes the errors, i.e. that maximizes the number of satisfied constraints.

We encode this problem as an LP problem with soft constraints. For each constraint $c_i \in Cs$ of the form $l_i > r_i$, we introduce a positive slack variable s_i and reformulate the constraint as $l_i + s_i \geq r_i + 1$. Our modified constraint set is denoted as Cs' , the set of all slack variables is SKs , and our objective function is the sum of the slack variables. The resulting linear program is:

$$\begin{aligned}
& \text{minimize} && \sum_{s \in SKs} s \\
& \text{subject to:} && Cs' \\
& && w_j \geq 0 \text{ for } h_j \in H^+ \\
& && w_j \leq 0 \text{ for } h_j \in H^- \\
& && s \geq 0 \text{ for } s \in SKs
\end{aligned}$$

This linear program can be efficiently solved by off-the-shelf solvers (we use Pulp [2]). Sect. 5 includes experiments that validate our weight inference approach.

5. Experimental Evaluation

In this section, we evaluate our disassembly algorithm and its corresponding weight inference algorithm on a large collection of binaries with ground truth. The evaluation is divided into four parts.

First, in Sect. 5.1, we experimentally validate our candidate generation algorithm (Sect. 3.2). Second, in Sect. 5.2 we evaluate our weight inference algorithm and how inferred weights generalize to unseen binaries. Third, in Sect. 5.3 we evaluate the overall disassembly effectiveness (precision and recall) and compare it to other state-of-the-art (SOTA) disassemblers. Fourth, in Sect. 5.4 we measure the runtime performance of both our improved disassembly algorithm, and the heuristic weight inference. All of our experiments are performed on stripped binaries.

Our experiments leverage four publicly available datasets⁷. Table 1 describes the Instruction Set Architecture

⁷ We only include the publicly available portion of Jiang’s (excluding SPEC binaries) and Assemblage’s PE dataset.

TABLE 1. EVALUATION DATASETS. EACH DATASET INCLUDES DIFFERENT ISAs AND BINARY FORMATS. EACH DATASET RELIES ON A DIFFERENT METHOD FOR EXTRACTING GROUND TRUTH.

Dataset	ISA	Format	Ground Truth
SOK [24]	x86/x64, arm32, aarch64	ELF	Compiler Modification
Pangine [19]	x86/x64	ELF	Intermediate Compilation Artifacts
Jiang [17]	arm32	ELF	Marker symbols
Assemb [20]	x86/x64	PE	PDB files

(ISA), binary formats, and ground truth source for each of the datasets. These datasets contain binaries compiled with a variety of compilers, including GCC, Clang, ICC, MSVC, and compiler optimizations O0-O3, Of, and Os. We refer the readers to the original publications for details.

Pang et al. [24] provide an extensive discussion on the limitations of each ground truth source. While modifying the compiler (as in dataset SOK) provides the highest quality ground truth, this is not feasible for closed-source compilers such as MSVC or ICC. The Pangine and Assemblage datasets (shortened as Assemb) complement Pang’s dataset with binaries compiled with those compilers.

During our experiments, we have identified limitations in the ground truth of each of the considered datasets. We provide a detailed account of these limitations in Appendix B.

We have released the code to run all the experiments⁸.

5.1. Candidate Block Generation

To substantiate the effectiveness of our algorithm presented in Sect. 3.2, we measure the prevalence of missed instructions and incorrect code block boundaries for each of the datasets. For each binary, we run our tool Ddisasm-WIS⁹ with a 1-hour timeout and collect information about the generated candidate blocks. We then compare that information with the ground truth to compute *missed instructions* and *incorrect code block boundaries* (see Sect. 3.2.3). A binary’s block candidate set is a sound overapproximation if it has neither, it is unsound otherwise. Thus, a binary can either be sound, unsound, or count as a failure. Failures include (1) timeouts (51 binaries), (2) Ddisasm-WIS failing to produce an output (3473 binaries), (3) failures in the ground truth extraction from PDBs (273 binaries), or (4) our scripts failing to decode an instruction marked as code by the ground truth (2545 binaries). The latter can be caused by limitations of our instruction decoder (Capstone 5.0.1 [27]), or it might signal an error in the ground truth.

Table 2 reports the number of binaries (and percentage) in each category, the total number of true instructions (Insns), the number (and percentage) of missed instructions, and the number of candidate blocks with incorrect boundaries (Incorrect Blocks). Our extended traversal (Sect. 3.2.2)

8. <https://github.com/GrammaTech/ddisasm-wis-evaluation>

9. Ddisasm-WIS’s improvements have been merged into the official Ddisasm repository <https://github.com/GrammaTech/ddisasm>. We run our experiments with commit 415a2be.

generates a sound overapproximation in the vast majority of cases. Consider SOK’s dataset. 6 out of the 7 unsound binaries in SOK’s x86/x64 dataset correspond to different versions of `libc` which present prefix-enclosed instructions (see Figure 1). While our tool can correctly handle those patterns, they are not currently handled by our evaluation scripts. As for SOK’s arm32 binaries, our experiments report 9 unsound binaries, all of them in the Thumb dataset. These binaries have 65 missing instructions where 61 of those are concentrated in 3 different versions of the same program (`ssh-keyscan`). We believe these are reported due to errors in the ground truth. The overall ratio of unsound binaries is less than 0.4% across all datasets, with the exception of Assemblage. We manually examined a sample of unsound binaries and discovered both inaccuracies in the PDB file, particularly with jump tables and obfuscated code, and some limitations of our tool. We provide additional details in Appendix C.

5.2. Weight Inference Evaluation

To evaluate our weight inference algorithm, we select a random sample of 1,895 binaries (approximately 20%) of our datasets SOK, Jiang, and Pangines as our training set. Our algorithm collects 86,564 constraints and produces an optimal weight assignment using Pulp [2]. The optimal weights satisfy 86,263 constraints, leaving 301 unsatisfied. We evaluate the resulting weights in practice by running Ddisasm-WIS with those weights on the complete dataset.

We compare the results against a baseline of Ddisasm-WIS with manually optimized weights. Note that Ddisasm-WIS’s manual weights are the result of more than four years of continuous development and thus we expect them to be close to optimal. For each binary, we measure the number of correctly recovered instructions (true positives), spurious instructions (false positives) and the number of true instructions that were not recovered (false negatives) and use them to compute the overall precision and recall. These metrics are computed for each dataset, ISA, training and validation subsets, for both manual and learned weights. The results can be found in Table 3. In addition, we report the number of correct binaries (binaries with no false positives nor false negatives) in each category.

Overall, learned weights achieve similar results as the manually tuned weights in most categories. We see improvements in the handling of SOK arm32 binaries where the recall increases from 99.906% to 99.973% (in the validation set), which also has an important effect on the number of correct binaries (from 35.77% to 44.06%).

Table 3 also demonstrates how weights generalize to unseen binaries. Despite training in only 20% of the binaries, all metrics are comparable on both the training and validation set, i.e. there is no overfitting, even in the Assemblage dataset, which was not part of the training set.

SOK’s arm32 results are surprising because Ddisasm-WIS’s recall in the training set is much lower than in the validation set (99.188% compared to 99.973%). Upon closer examination, we noticed this is due to a single

TABLE 2. DDISASM CANDIDATE BLOCK GENERATION EVALUATION.

Dataset	ISA	# Binaries	Failures (%)	Sound (%)	Unsound (%)	Insns	Missed Insns (%)	Incorrect Blocks
SOK	x86/x64	3,974	113 (2.84%)	3,854 (96.98%)	7 (0.18%)	179,630,938	22 (0.00001%)	0
SOK	arm32	2,561	49 (1.91%)	2,503 (97.74%)	9 (0.35%)	89,081,800	65 (0.00007%)	2
SOK	aarch64	1,264	0 (0%)	1,264 (100%)	0 (0%)	45,484,081	0 (0%)	0
Pangine	x86/x64	879	61 (6.94%)	816 (92.83%)	2 (0.23%)	104,427,721	0 (0%)	2
Jiang	arm32	1,026	0 (0.00%)	1,025 (99.90%)	1 (0.10%)	30,297,775	4 (0.00001%)	0
Assemb	x86/x64	85,296	6,119 (7.17%)	78,659 (92.22%)	518 (0.61%)	1,458,731,926	5,396 (0.00037%)	90

TABLE 3. WEIGHT INFERENCE EVALUATION: NUMBER OF CORRECT BINARIES, PRECISION, AND RECALL FOR BOTH THE TRAINING AND VALIDATION SETS, USING BOTH MANUALLY OPTIMIZED WEIGHTS AND LEARNED WEIGHTS.

Dataset	ISA	Method	Training Set				Validations Set			
			# Binaries	Correct (%)	Precision	Recall	# Binaries	Correct (%)	Precision	Recall
SOK	x86/x64	Manual	738	687 (93.09%)	100.000%	99.995%	3,236	2,968 (91.72%)	99.991%	99.997%
		Learned		694 (94.04%)	100.000%	99.995%		2,993 (92.49%)	99.991%	99.997%
SOK	arm32	Manual	534	199 (37.27%)	99.965%	99.120%	2,027	725 (35.77%)	99.954%	99.906%
		Learned		244 (45.69%)	99.964%	99.188%		893 (44.06%)	99.952%	99.973%
SOK	aarch64	Manual	253	252 (99.60%)	100.000%	100.000%	1,011	1,009 (99.80%)	100.000%	100.000%
		Learned		252 (99.60%)	100.000%	100.000%		1,008 (99.70%)	100.000%	100.000%
Pangine	x86/x64	Manual	171	163 (95.32%)	100.000%	100.000%	708	658 (92.94%)	100.000%	99.999%
		Learned		163 (95.32%)	100.000%	100.000%		645 (91.10%)	100.000%	99.999%
Jiang	arm32	Manual	199	148 (74.37%)	99.988%	99.995%	827	635 (76.78%)	99.994%	99.995%
		Learned		153 (76.88%)	99.988%	99.996%		637 (77.03%)	99.994%	99.996%
Assemb	x86/x64	Manual					85,296	77,186 (90.49%)	99.960%	99.996%
		Learned						76,613 (89.82%)	99.960%	99.994%

binary `perlbench_base.arm32-gcc81-0s` that has 150,849 false negatives (over 99% of all the false negatives). After excluding it, the resulting overall recall is 99.993%.

Despite their similar performance, learned weights set 46 (out of 95) heuristics to 0. Four of those are heuristics reliant on symbols, which produce no matches on stripped binaries. This leaves a total of 42 heuristics (> 40% of all heuristics) that can be removed without a performance impact. Fewer heuristics simplify development and code maintenance, making the weight inference algorithm a powerful tool to evaluate heuristics and measure their importance. For example, Ddisasm-WIS has two heuristics that assign points when a candidate block is called or jumped to by another candidate block respectively. When learning optimal weights, the heuristic that matches jumps gets zero weight whereas the one that detects calls has a non-zero weight. This agrees with Priyadarshan et al.’s findings that spurious (short) jumps are much more common than spurious calls in x64 [26].

5.3. Comparison to Other Disassemblers

We evaluate Ddisasm-WIS’s disassembly effectiveness by comparing it to other disassemblers. SOK’s dataset contains disassembly information for several state-of-the-art tools, namely Binary Ninja, IDAPro, Ghidra, and Angr. We include these results together with ours in Table 4. For Ddisasm-WIS, we use the learned weights from Sect. 5.2. In addition, we compare Ddisasm-WIS to an older version of Ddisasm without interval scheduling (Ddisasm-1.6), and to two recent disassemblers: D-ARM [32] (commit 40b5462) on the arm32 datasets (Jiang and SOK), and DASSA [26] on x64.

Ddisasm-WIS obtains the highest recall of all the tools on all datasets. At the same time, it is second best in precision for SOKs’ x86/x64 and arm32 datasets. In all cases, Ddisasm-WIS’s precision is within 0.01% of the highest precision. Ddisasm-WIS achieves the highest rates of correct disassembly in all categories, which is a crucial metric for binary rewriting applications.

Our Ddisasm-WIS version improves over Ddisasm-1.6 in all the arm32 metrics (e.g., 99.99% vs. 99.92% on the Jiang dataset) except for SOK arm32’s recall, which is lower due to the outlier discussed in the previous experiment (`perlbench_base.arm32-gcc81-0s`). The improvements are more visible when considering the percentage of correct binaries (binaries with perfect disassembly). However, Ddisasm-WIS suffers a small performance regression for x86/x64. Nevertheless, our version has fewer failures in that dataset as well (0.96% vs. 2.44%).

DASSA’s evaluation reports different metrics, but their artifact evaluation also collects precision and recall. These metrics also differ slightly due to how they are computed. Note that we aggregate all true positives, false positives, and false negatives across all binaries and compute an overall precision and recall, whereas they compute precision and recall for each binary and then compute the mean. Our approach gives more weight to larger binaries that contain more instructions rather than averaging over binaries of very different size. Finally, in DASSA’s evaluation, DASSA outperforms Ddisasm (the version is not specified) whereas in our evaluation both Ddisasm-WIS and Ddisasm-1.6 yield better results than DASSA. The discrepancy is likely due to DASSA’s evaluation removing exception information from binaries, which could affect Ddisasm’s accuracy. Exception information is used (when available) for the candidate gen-

TABLE 4. TOOL COMPARISON: PERCENTAGE OF FAILURES, CORRECTLY DISASSEMBLED BINARIES, PRECISION, AND RECALL FOR EACH DATASET AND EACH TOOL.

Dataset ISA	Tool	Fail or Missing	Correct	Precision	Recall
SOK x86/x64	Ddisasm-WIS	0.96%	92.8%	99.992%	99.996%
	Ddisasm-1.6	2.44%	94.9%	99.993%	99.998%
	DASSA	*49.72%	*33.6%	99.976%	99.948%
	Ninja	8.76%	12.6%	99.983%	97.151%
	Ida	0.03%	69.1%	99.994%	99.945%
	Ghidra	20.56%	43.0%	99.789%	93.635%
	Angr	17.89%	60.3%	99.908%	99.984%
SOK arm32 (thumb)	Ddisasm-WIS	0.00%	37.3%	99.905%	99.968%
	Ddisasm-1.6	7.09%	1.9%	99.869%	99.848%
	D-ARM	0.00%	0.1%	97.733%	95.382%
	Ninja	0.00%	0.8%	98.577%	91.987%
	Ida	0.00%	8.9%	99.391%	96.388%
	Ghidra	0.00%	1.4%	99.913%	89.046%
	Angr	4.14%	0.1%	97.365%	98.682%
SOK arm32	Ddisasm-WIS	0.08%	51.3%	99.996%	99.705%
	Ddisasm-1.6	12.72%	28.4%	99.993%	99.765%
	D-ARM	0.00%	0.0%	92.582%	91.927%
	Ninja	0.00%	21.7%	99.993%	96.551%
	Ida	0.00%	11.5%	99.998%	97.955%
	Ghidra	0.00%	1.1%	99.969%	94.681%
	Angr	4.37%	15.9%	98.524%	99.313%
SOK aarch64	Ddisasm-WIS	0.00%	99.7%	100.000%	100.000%
	Ddisasm-1.6	0.00%	99.7%	100.000%	100.000%
	Ninja	1.19%	2.9%	100.000%	97.274%
	Ida	3.16%	0.0%	100.000%	91.172%
	Ghidra	1.19%	41.1%	100.000%	97.505%
	Angr	1.66%	96.8%	100.000%	100.000%
Jiang arm32	Ddisasm-WIS	0.00%	77.0%	99.993%	99.996%
	Ddisasm-1.6	0.68%	17.6%	99.985%	99.929%
	D-ARM	0.00%	1.2%	96.754%	92.249%

*DASSA does not support x86 32bit. It correctly disassembles 67.6% of the x64 binaries.

eration traversals (see Sect. 3.2.1)

Despite our best efforts, we could not reproduce D-ARM’s reported results [32]. In particular, its authors evaluated D-ARM against the AOSP dataset—a subset of Jiang’s dataset consisting of 669 Android libraries. Their reported precision and recall on AOSP are 99.79% and 99.86%. In contrast, our experiments yield 97.09% and 92.42% for D-ARM. Note that Ddisasm-WIS achieves 99.992% precision and 99.996% recall for AOSP, which is well above both the reported and measured D-ARM rates.

5.4. Runtime Performance

We conduct two sets of runtime measurements to evaluate the performance of Ddisasm-WIS. First, in Section 5.4.1, we demonstrate the practicality of Ddisasm-WIS’s extended traversal and conflict resolution algorithm by comparing its runtime against Ddisasm-1.6. Second, in Section 5.4.2, we assess the overhead introduced by collecting and solving constraints for weight inference.

5.4.1. Ddisasm-WIS vs. Ddisasm-1.6. Table 5 shows the comparison of the total runtimes of Ddisasm-WIS against Ddisasm-1.6. In all datasets except SOK x64/x64, Ddisasm-WIS is slightly slower than Ddisasm-1.6 (at most 28% slower in the worst performing category aarch64). While

TABLE 5. DDISASM-WIS VS. DDISASM-1.6 RUNTIME COMPARISON: NUMBER OF BINARIES THAT BOTH TOOLS SUCCEEDED ON, TOTAL RUNTIMES, AND DDISASM-WIS RUNTIME RELATIVE TO DDISASM-1.6.

Dataset ISA	# Common Binaries	Runtime (s)		Ratio WIS/1.6
		Ddisasm-WIS	Ddisasm-1.6	
SOK x86/x64	3,868	165,513	191,850	86%
SOK arm32 (thumb)	1,167	37,702	33,437	112%
SOK arm32	1,138	42,694	40,535	105%
SOK aarch64	1,264	21,458	16,735	128%
Jiang arm32	1,019	20,651	16,512	125%
Pangine x86/x64	839	137,862	124,103	111%

TABLE 6. CONSTRAINT INFERENCE RUNTIME: NUMBER OF BINARIES, TOTAL CONSTRAINT INFERENCE (CI) RUNTIME, DISASSEMBLY RUNTIME (D), AND CI RUNTIME RELATIVE TO DISASSEMBLY.

Dataset ISA	# Binaries	Runtime (s)		Ratio CI / D
		CI	D	
SOK x86/x64	738	4,909	22,542	21%
SOK arm32 (thumb)	260	1,440	7,205	19%
SOK arm32	274	13,284	8,509	156%
SOK aarch64	253	572	3,746	15%
Jiang arm32	199	2,340	4,661	50%
Pangine x86/x64	171	5,918	15,701	37%

the comparison does not isolate all the factors that determine this performance difference, it demonstrates that Ddisasm-WIS’s improved disassembly algorithm is practical. Given the improved results of our tool, we believe that this runtime increase is acceptable.

5.4.2. Weight Inference. The results in Table 6 highlight the runtime overhead introduced by collecting weight constraints, offering insight into the efficiency of the weight-inference step in our algorithm. In five out of six datasets, excluding SOK arm32, weight inference completes significantly faster than the overall disassembly process, often taking less than 50% of the time total runtime. The only outlier is the SOK arm32 dataset, where weight inference takes longer than disassembly. In this dataset, the disassembly algorithm creates large candidate data blocks representing jump tables, which prevents effective interval scheduling decomposition (see Sect. 4.3.1), thereby harming the constraint inference performance. Nonetheless, since weight inference is a one-time preprocessing step whose results are leveraged by Ddisasm-WIS, we argue that its overhead remains acceptable. Pulp solved the LP problem with the 86,263 collected constraints in approximately 10 seconds.

6. Related Work

Our work focuses on static disassembly and, in particular, on instruction recovery. It extends Ddisasm [15] with an extended candidate block generation to handle arm32 and aarch64 binaries, and a novel conflict resolution algorithm. In contrast to the original Ddisasm, which only considered overlaps between two blocks at a time, our conflict resolution algorithm is expressed as a weighted interval scheduling (WIS) problem that jointly considers all block overlaps.

D-ARM’s [32] is a closely related approach. We note three important differences. First, D-ARM is tailored for arm and aarch64 binaries, whereas our approach performs well for x86, and x64 as well. Second, D-ARM does not have a candidate generation phase. D-ARM instead considers all individual instructions (superset disassembly) as candidates. Third, D-ARM’s conflict resolution is encoded as a maximum weight independent set (MWIS) optimization problem. While WIS only considers overlaps, MWIS can enforce control flow dependencies as well. For example, if a candidate code block A has a call to another candidate code block B , then A implies B . These dependencies make the optimization problem NP-Hard, which dictates the adoption of greedy approximation methods.

As we have argued in Sect. 3, grouping instructions into blocks is especially important for x86 and x64, because it reduces the number of block candidates by an order of magnitude compared to superset instructions. We believe that our candidate generation phase and control flow-based heuristics compensate for the lack of dependencies in the conflict resolution phase. Considering multi-instruction candidate blocks effectively enforces dependencies among instructions in the same code block. Our experiments indicate that WIS is sufficient, given an appropriate set of heuristics. However, further research should be conducted to understand the tradeoffs of both approaches better.

Priyadarshan et al. [26] recently presented the DASSA disassembler. DASSA is focused on x64. It computes a superset of all possible code (akin to our candidate generation) and also implements a conflict resolution method. DASSA’s conflict resolution is greedy. Each step considers a single candidate with the highest score (weight). For each candidate, DASSA computes a score based on statistical properties of the data (DASSA refines the probabilistic analysis of previous works [21]), and several static analysis-based checks to flag invalid code. Whether a candidate is selected depends on a combination of both the score and the checks. In contrast, our approach encodes both statistical properties and static analysis-based checks into a single weight. This results in a simpler conflict resolution algorithm. This integration is facilitated by our weight inference algorithm, which automatically adjusts the relative weights of the different heuristics. Integrating DASSA’s analyses and checks into our approach could further improve its accuracy.

Recent works attempt to leverage deep learning for disassembly, notably XDA [25] and DeepDi [33]. XDA analyzes raw bytes directly using a specially designed language model. DeepDi performs superset disassembly [7],

and embeds instruction using a graph neural network that captures dependencies among instructions. Both D-ARM and DASSA’s evaluations compare against and outperform XDA. DeepDi’s evaluation shows that Ddisasm (the version is not specified) achieves higher accuracy than DeepDi.

Traditional disassemblers also combine static analyses and heuristics. Pang et al. [23] provides a detailed overview for x86/x64 and Jiang et al. [17] perform an extensive comparative of arm32 disassemblers.

Most static disassemblers [1], [3], [6], [8], [15], [16], [22], [29] recover additional information, such as control flow graphs, function boundaries [4], symbolization information [11], [30], [31], or even perform decompilation [9]. Disassembly can also be performed dynamically, or through a combination of static and dynamic techniques [34]. Despite our significant progress (see Table 4), binary rewriting applications have very low tolerance for errors. Thus, multiple approaches have been proposed to instrument programs without requiring perfect disassembly [7], [10], [12], usually at the expense of some performance overhead.

7. Conclusion

In this paper we present a novel three-phase disassembly algorithm: (1) a candidate generation phase produces an overapproximation of all code blocks; (2) a weight assignment phase assigns a weight to each candidate block; and (3) a conflict resolution phase resolves overlaps producing the final set of blocks. Conflict resolution is expressed as a WIS problem, which has an efficient optimal solution.

We also present a solution to the weight assignment problem, i.e. how to optimally assign weights to heuristics to maximize accuracy. Our solution optimizes heuristic weights based on binaries annotated with ground truth. For each binary, it collects a set of linear constraints over unknown heuristic weights that ensure correct disassembly. We maximize the number of satisfied constraints from all the binaries by solving a LP problem with soft constraints.

We have implemented our approach on top of Ddisasm [15] and performed a large experimental evaluation to validate it. Our approach supports multiple ISAs (x64, x86, aarch64, and arm32) and outperforms (in number of correct binaries and recall) or nearly matches (with less than 0.002% difference in precision) SOTA disassemblers. Our weight inference algorithm results in weights that outperform manually tuned weights in several benchmarks, while effectively reducing the number of heuristics (by setting their weights to 0) by 40%.

References

- [1] Hex-rays: The IDA Pro disassembler and debugger. <https://www.hex-rays.com/products/ida/>.
- [2] Pulp. <https://coin-or.github.io/pulp/>.
- [3] National Security Agency. Ghidra, 2019. <https://www.nsa.gov/resources/everyone/ghidra/>.
- [4] D. Andriess, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 177–189, April 2017.

- [5] Dennis Andriess, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *The 25th USENIX Security Symposium*, pages 583–600, Austin, TX, 2016. USENIX Association.
- [6] Cryptic Apps. Hopper. <https://www.hopperapp.com/>.
- [7] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 01 2018.
- [8] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 463–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [9] Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Evaluating the effectiveness of decompilers. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 491–502, New York, NY, USA, 2024. Association for Computing Machinery.
- [10] Luca Di Bartolomeo, Hossein Moghaddas, and Mathias Payer. Armore: pushing love back into binaries. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC ’23, USA, 2023. USENIX Association.
- [11] Sushant Dinesh, Nathan Burrow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *The 41st Symposium on Security and Privacy*. IEEE, 2020.
- [12] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 151–163, New York, NY, USA, 2020. Association for Computing Machinery.
- [13] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World’s Most Popular Disassembler*. No Starch Press, 2011.
- [14] Daniel Engel, Freek Verbeek, and Binoy Ravindran. On the decidability of disassembling binaries. In *Theoretical Aspects of Software Engineering: 18th International Symposium, TASE 2024, Guiyang, China, July 29 – August 1, 2024, Proceedings*, page 127–145, Berlin, Heidelberg, 2024. Springer-Verlag.
- [15] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1075–1092. USENIX Association, August 2020.
- [16] Vector 35 Inc. Binary ninja: a new kind of reversing platform. <https://binary.ninja/>.
- [17] Muhui Jiang, Qinming Dai, Wenlong Zhang, Rui Chang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. A comprehensive study on arm disassembly tools. *IEEE Transactions on Software Engineering*, 49(4):1683–1703, 2023.
- [18] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA, 2005.
- [19] Kaiyuan Li, Maverick Woo, and Limin Jia. On the generation of disassembly ground truth and the evaluation of disassemblers. In *Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation*, FEAST’20, page 9–14, New York, NY, USA, 2020. Association for Computing Machinery.
- [20] Chang Liu, Rebecca Saul, Yihao Sun, Edward Raff, Maya Fuchs, Townsend Southard Pantano, James Holt, and Kristopher Micinski. Assemblage: Automatic binary dataset construction for machine learning. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.
- [21] Kenneth Miller, Yonghui Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *International Conference on Software Engineering (ICSE)*. ACM, 2019.
- [22] pancake. radare. <https://www.radare.org/rl/>.
- [23] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *42nd IEEE Symposium on Security and Privacy (SP)*, 2021.
- [24] Chengbin Pang, Tiantai Zhang, Ruotong Yu, Bing Mao, and Jun Xu. Ground truth for binary disassembly is not easy. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2479–2495, Boston, MA, August 2022. USENIX Association.
- [25] Kexin Pei, Jonas Guan, David Williams King, Junfeng Yang, and Suman Jana. Xda: Accurate, robust disassembly with transfer learning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*, 2021.
- [26] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. Accurate disassembly of complex binaries without use of compiler metadata. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS ’23, page 1–18, New York, NY, USA, 2024. Association for Computing Machinery.
- [27] Nguyen Anh Quynh. Capstone: Next-gen disassembly framework. *Black Hat USA*, 2014.
- [28] Shuichi Sakai, Mitsunori Togasaki, and Koichi Yamazaki. A note on greedy algorithms for the maximum weighted independent set problem. *Discrete Applied Mathematics*, 126(2):313–322, 2003.
- [29] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, May 2016.
- [30] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.
- [31] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, Washington, D.C., 2015. USENIX Association.
- [32] Yapeng Ye, Zhuo Zhang, Qingkai Shi, Youssa Aafer, and Xiangyu Zhang. D-arm: Disassembling arm binaries by lightweight superset instruction interpretation and graph modeling. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2391–2408, 2023.
- [33] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. DeepDi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2709–2725, Boston, MA, August 2022. USENIX Association.
- [34] Zhuo Zhang, Wei You, Guan hong Tao, Youssa Aafer, Xuwei Liu, and X. Zhang. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. *2021 IEEE Symposium on Security and Privacy (SP)*, pages 659–676, 2021.

Appendix A. Proofs

This section contains the proofs for Theorem 1 and Theorem 2. For convenience, we restate the theorems before proceeding with the proof.

Theorem 1. *If a candidate block set $CBlocks$ does not present missed instructions nor incorrect code block boundaries, then $CBlocks$ is a sound overapproximation.*

Proof of Theorem 1. We prove that we can build a subset $Blocks \in CBlocks$ such that $\bigcup_{b \in Blocks} Insns(b) = Code$. Our proof relies on the fact that candidate code blocks do not share instructions (See Sect. 3.2.1). If $CBlocks$ has no

missed instructions, every $i \in Code$ belongs to a single block $b \in CBlocks$ and $BlockOf(i)$ (see Definition 3) yields a single block for each $i \in Code$. Thus, we can define our final block set $Blocks = \bigcup_{i \in Code} BlockOf(i)$. By construction $\bigcup_{b \in Blocks} Insns(b) \supseteq Code$.

Next, we prove that $\bigcup_{b \in Blocks} Insns(b) \subseteq Code$, which implies that $\bigcup_{b \in Blocks} Insns(b) = Code$ and consequently $CBlocks$ is a sound overapproximation. If $CBlocks$ has no incorrect code block boundaries, for every block $b \in CBlocks$ we have either $Insns(b) \cap Code = \emptyset$ or $Insns(b) \subseteq Code$. Since we have defined our set $Blocks$ using $BlockOf$, we know the first condition does not hold. Therefore, for every $b \in Blocks$ we know that $Insns(b) \subseteq Code$, which implies $\bigcup_{b \in Blocks} Insns(b) \subseteq Code$. \square

Theorem 2. *Let Cs be the set of constraints generated by Algorithm 2 and let $\alpha : H \rightarrow \mathbb{Z}$ be a weight assignment that satisfies the constraints $\alpha \models Cs$, then there is an optimal schedule S such that $\alpha(SW(S)) > \alpha(SW(S'))$ for every non-optimal schedule S' .*

Proof of Theorem 2. First, we prove that at the end of the loop, $symopt[n]$ contains tuples representing all possible schedules with non-overlapping blocks. The same argument that ensures the correctness of WIS algorithm (Algorithm 1) guarantees can be applied here.

We prove it by complete induction over i . Our inductive invariant is that at each iteration i , $symopt[i]$ contains tuples for all schedules including blocks from b_1 to b_i . The base case ($i=0$) is trivial with an empty schedule, and the inductive step i includes all possible schedules with b_i (*take*) and all the possible schedules without b_i (*leave*).

Since $symopt[n]$ contains all the possible schedules, $bestSched$ will return an optimal schedule S . Line 10 collects all non-optimal schedules by relying on Lemma 1, and Line 11 generates a constraint $SW(S) > SW(S')$ for each non-optimal schedule. Thus, for each non-optimal schedule S' there is a constraint $SW(S) > SW(S')$ in Cs . Since $\alpha \models Cs$ it satisfies $\alpha \models SW(S) > SW(S')$ as well. \square

Appendix B. Ground Truth Limitations

During our experiments, we have identified multiple limitations in the ground truth provided by the different dataset (see Table 1). In this section, we detail our findings and how we address them in our experiments.

B.1. SOK ground truth limitations

SOK’s ground truth does not distinguish ARM and Thumb decode modes, instead it has two separate datasets for each mode. Unfortunately, we have observed that some Thumb binaries contain a few ARM functions, usually added by the linker. SOK’s evaluation tries to exclude functions added by the linker (we add those to *Ignored*), but it fails if the binaries are stripped before collecting the ground truth¹⁰. Binaries can be recompiled to avoid stripped binaries,

10. <https://github.com/junxzm1990/x86-sok/issues/32>

but that would also require rerunning all the disassemblers on the newly compiled binaries, since small changes in the compilation environment could lead to differences in the generated binaries.

Instead, we reuse the originally provided datasets and we detect ARM functions in Thumb binaries by checking the consistency of the reported instruction sizes. If the ground truth reports an instruction size of 4 at address and the instruction decoder returns a instruction of size 2 at that same address using Thumb mode, then that instruction (and its corresponding function) must be in ARM decode mode. Note that naively looking for 4 byte instructions is not enough because some some Thumb instructions are indeed 4 bytes long. Note also that this method might not find all ARM functions in Thumb binaries, which could introduce some small errors in the arm32 Thumb evaluation.

During our experiments, we have also identified other errors in SOK’s ground truth, which we have reported¹¹. These errors were cause by (1) a failure to correctly handle duplicated sections during linking, (2) an unsupported instruction `udf` in the compiler instrumentation that recovers the ground truth. Both issues have been promptly resolved by the authors, but the datasets have not been re-generated. We exclude manually excluded the detected cases of (1) and extended our evaluation scripts to recover the missing `udf` from the ground truth.

B.2. Panguine ground truth limitation

This ground truth is limited to regions within well-defined function boundaries, and thus it is incomplete. We add the regions outside function boundaries to *Ignored*. In addition, we have also found and reported issues with Panguine’s ground truth¹². Some MSVC-compiled binaries contain jump tables being classified as code. These are classified as *optional* true instructions, which indicates a lower degree of certainty in the ground truth. We followed the authors’ recommendation and ignored optional true instructions in our evaluation scripts.

B.3. Jiang ground truth limitations

Jiang’s arm32 dataset relies on compiler-generated marker symbols: \$a, \$t, and \$d for ARM, Thumb, and data respectively for extracting ground truth information.

Unfortunately, marker symbols are not always accurate on binary regions added by the linker. We account for this limitation by considering marker symbols only within the boundaries of function symbols (other regions are added to *Ignored*). We have found this restriction to yield more reliable ground truth at the expense of completeness (larger parts of the binaries are ignored). In addition, we have also semi-automatically annotated some of those ignored regions that contain specific patterns, such as ARM/Thumb interworking veneers, that can be matched with high confidence.

11. <https://github.com/junxzm1990/x86-sok/issues/31>

12. <https://github.com/panguine/disasm-benchmark/issues/2>

TABLE 7. MANUAL ASSESSMENT OF UNSOUND PE BINARIES

Category	Affected Binaries	Issue Description	Impact on Disassembly
Missing Jump-Table Information	SymbolLoadDLL_d.dll, ChisMath.exe, DrawDemo.exe, GetScreenRGB.exe, osu-memory-demo.exe, threadpool.exe	PDB files lack information about jump tables, leading to incomplete or incorrect function boundaries.	Our tool correctly identifies jump tables, leading to discrepancies compared to PDB ground truth.
Misinterpreted Padding Bytes	MaiSense.dll, iphubliclient_amxx.dll, gtest_unittest.exe, SysInv32.exe, ProfilerOBJ.dll, Test.exe	A <code>jmp</code> instruction targets a sequence of <code>int 3</code> instructions, typically used as padding.	Our tool treats <code>int 3</code> sequences as padding rather than executable code, marking the <code>jmp</code> instruction targeting them as data as well.
Obfuscated Code	hovew-crackme.exe, SPASM.exe	Unusual control-flow structures and inline data embedded within code segments result in incorrect PDB information.	PDB files misrepresent function boundaries, while our tool correctly handles the obfuscated code.
Ddisasm-WIS bug	psp.exe	A minor bug causes a call instruction target to be computed incorrectly, which results in the instruction invalidation.	Future work will address this case.

Finally, we have completely excluded 7 binaries where we found incorrect ground truth. These are mostly binaries implementing cryptographic primitives (such as `libcrypto`) where developers have encoded instructions as data directly (resulting in incorrect marker symbols).

B.4. Assemblage ground truth limitations

While PDB files are a useful source of ground truth for evaluating disassembly, they are not entirely reliable, particularly in cases involving jump tables and obfuscation. See Appendix C for more details.

Appendix C. Manual Assessment of Unsound PE Binaries

Our candidate generation evaluation (Sect. 5.1), results in a significantly higher ratio of unsound binaries for the Assemblage PE binaries (0.61% vs. 0.35% for second worst category). To investigate further, we randomly selected 15 binaries from the 518 unsound binaries (Table 2) and manually inspected them. The results of this analysis can be found in Table 7. We find three primary causes for binaries being classified as unsound: missing jump-table information in PDB files, misinterpretation of padding bytes, and obfuscated code¹³.

C.1. Missing Jump-Table Information

We found six binaries with missing jump-table information in the PDB debug data. Jump tables are commonly used for indirect branching, typically within the code section of PE binaries. Our tool successfully reconstructed these jump tables, correctly identifying function boundaries and indirect branch targets. The discrepancy between our generated candidates and the PDB ground truth is, therefore, due to a limitation of the PDB files.

¹³. Given the limited size of the sample, additional types of issues may exist, and the relative frequency of each category may differ in the full dataset.

C.2. Misinterpreted Padding Bytes

MSVC binaries use `int 3` (software breakpoint) instructions as padding (to align function boundaries). While having a single `int 3` instruction as part of the executable code is possible, a sequence of *consecutive* `int 3` instructions is very unlikely to be meant for execution. In other words, sequences of `int 3` instructions are indeed very common in PE binaries, but they are never executed. Thus, Ddisasm-WIS considers sequences of `int 3` as invalid. This invalidation is then propagated backwards to instructions jumping to those sequences directly during the first disassembly traversal (see backward traversal in Section 3.2.1). We find instructions in the PDB ground truth being invalidated because of this reason in six binaries. We believe these jumps to `int 3` sequences might be a case of obfuscation. Unfortunately, the PDB ground truth does not explicitly clarify whether such `int 3` sequences should be considered executable (since the PDB ground truth is incomplete), making it challenging to establish a definitive interpretation.

C.3. Obfuscated Code

In two instances, we found obfuscated code patterns designed to confuse disassembly tools. For example, we found conditional jumps targeting the middle of multi-byte instructions or raw byte sequences (not decodable as instructions), which result in ambiguous control flow and overlapping instruction boundaries. Other examples included manually inserted bytes that disrupted typical linear disassembly. In these cases, the PDB files provided incorrect instruction boundaries, whereas our tool, leveraging more heuristics, correctly disassembled the obfuscated code.

C.4. Other Cases

The remaining unsound binary, `psp.exe`, is caused by a bug in our tool. We plan to address this bug shortly.