D-ARM: Disassembling ARM Binaries by Lightweight Superset Instruction Interpretation and Graph Modeling

Yapeng Ye†, Zhuo Zhang†, Qingkai Shi†, Yousra Aafer‡, Xiangyu Zhang†
†Purdue University, ‡University of Waterloo
{ye203, zhan3299, shi553}@purdue.edu, yousra.aafer@uwaterloo.ca, xyzhang@cs.purdue.edu

***Abstract*—ARM binary analysis has a wide range of applications in ARM system security. A fundamental challenge is ARM disassembly. ARM, particularly AArch32, has a number of unique features making disassembly distinct from x86 disassembly, such as the mixing of ARM and Thumb instruction modes, implicit mode switching within an application, and more prevalent use of inlined data. Existing techniques cannot achieve high accuracy when binaries become complex and have undergone obfuscation. We propose a novel ARM binary disassembly technique that is particularly designed to address challenges in legacy code for 32-bit ARM binaries. It features a lightweight superset instruction interpretation method to derive rich semantic information and a graph-theory based method that aggregates such information to produce final results. Our comparative evaluation with a number of state-of-the-art disassemblers, including Ghidra, IDA, P-Disasm, XDA, and D-Disam, on thousands of binaries generated from SPEC2000 and SPEC2006 with various settings, and real-world applications collected online show that our technique** D-ARM **substantially outperforms the baselines.**

I. Introduction

ARM is one of the two most popular architecture families (the other is x86). Since it was introduced in 1980s, it gradually prevails due to its excellent balance between performance and resource consumption. It dominates in mobile computing as 95% of high-end smartphones are based on ARM [1] and is also widely used in high performance computing [2]. With the extreme popularity of ARM devices, securing ARM applications on such devices is hence critical. Just like x86 binary analysis is a key enabling technique for x86 software security, ARM binary analysis is critical for ARM application security. It serves a large number of downstream applications such as ARM malware analysis [3], [4], ARM application fuzzing [5], [6], ARM rewriting for application hardening [7], [8], and ARM code debloating that reduces the footprint of an application to achieve a smaller attack surface [9], [10].

A fundamental challenge in ARM binary analysis is ARM disassembly, which is the critical first step of any binary analysis. Compared to x86, ARM, particularly AArch32, poses a number of unique challenges. For example, it supports two instruction modes in a same 32-bit application, ARM and Thumb, with the former high performance and the latter compact. In other words, some instructions in a binary are in ARM and the others are in Thumb. The mode switching may be implicit, depending on values computed in registers at runtime. It can occur anywhere and any time. The byte sequence starting at an address can be decoded to different instructions depending on the mode. In addition, inlined data are more prevalent in both 32-bit and 64-bit ARM binaries. ARM/Thumb instructions have fixed lengths that are much smaller than x86 instructions. They hence have limited room to encode immediate values (i.e., constants). As a result, such values are often present in memory as inlined data. Disassemblers may have difficulty recognizing them and falsely disassemble them to instructions.

There is a body of mature and effective works for x86 disassembly [11], [12], [13], [14], [15], [16]. However, they have limited effectiveness when applied to ARM binaries. The traditional linear sweeping disassemblers such as *objdump* [16] and recursive control flow traversal based disassemblers, such as *IDA* [11] and *Ghidra* [12], have difficulties handling inlined data, implicit mode switching, and indirect control flow transfer. *P-Disasm* [14] considers that each address may potentially start an instruction and uses a probabilistic method to compute posterior probabilities of addresses denoting true instructions. The probabilities are computed based on a number of hints such as the number of definition-use relations encountered. However, as shown by our results (Section VI), the probabilities can hardly be used to correctly decide if an instruction should be ARM or Thumb. Machine learning based disassemblers such as *XDA* [13] learn how to disassemble a binary by training on a huge set of samples. However, they largely depend on the syntactic patterns in these samples whereas implicit instruction mode switching is a complex semantic property. As such, they may not work well when binaries become complex (e.g., with obfuscation). Its F1 score on obfuscated binaries can be as low as 2.83% (Section VI).

In this paper, we propose a novel technique for ARM binary disassembly. We observe that a key challenge of recognizing instruction mode switching can hardly be handled by syntax based analysis. In addition, ARM code heavily uses load and store instructions, causing substantial memory aliasing and rendering simple local program analysis ineffective. We hence propose a lightweight static analysis to collect rich semantic information. The static analysis works on a *superset of instructions*, meaning that it disregards the true mode for a code address, which is unknown, and decodes the address to instructions in both modes and interprets them accordingly

in an abstract domain. Here, we call the multiple instructions (in different modes) that can be legally decoded at an address *the superset instructions* for the address. The semantic information derived by the static analysis (e.g., register, memory dependencies, and indirect control flow) is further leveraged by a phase of graph analysis to produce the final results. We model the program as a graph, with multiple nodes created for each address, denoting that the address shall be considered as an ARM instruction, a Thumb instruction, and data bytes, respectively. Edges are introduced between these nodes to denote their inter-constraints based on the analysis results, such as an address cannot be an ARM instruction and a Thumb instruction at the same time and an ARM instruction must be followed by another ARM instruction if there is no mode switching. Then the disassembly problem is reduced to a *maximum weight independent set* (MWIS) problem [17] that maximizes the total weight while respecting the constraints. Our contributions are summarized as follows.

- We develop a disassembly technique for ARM binaries. It features a novel lightweight static analysis technique that can interpret a binary without even knowing how to correctly disassemble the binary.
- We also propose a novel graph modeling method for ARM binaries. We formally prove that with the graph model, we can reduce the ARM disassembly problem to an MWIS problem, which is NP-hard. An existing approximate solution then can be leveraged to derive the final results.
- We implement a prototype D-ARM. We evaluate it on more than 5000 binaries, including those compiled from SPEC2000 and SPEC2006 with different compilation options, architecture and instruction set configurations, those collected online, and those undergone obfuscation. We compare it with five baselines: Ghidra [12], IDA [11], P-Disasm [14], XDA [13], and D-Disasm [15]. Our results show that D-ARM is almost always the best performing tool, with an F1 score higher than 95% in most cases. In the presence of substantial obfuscation, the other tools have only 2.83-56.45% F1 scores whereas D-ARM still has 78.16-88.72%. Our case study also shows that D-ARM can benefit downstream binary rewriting with fewer execution failures and incorrect coverage reports. D-ARM is publicly available [18].

II. Background

ARM is a family of Reduced Instruction Set Computing (RISC) architectures, which are quite different from the Complex Instruction Set Computing (CISC) architectures such as the Intel x86 families. In this section, we introduce the main features of ARM that may affect its disassembly.

A. Multiple Instruction Sets

There have been several generations of ARM architectures in the past decades. Different architectures have different features and support mixed instruction sets. The early ARM versions are of 32-bit architecture (AArch32) and only use the 32-bit ARM instruction set (A32). Since ARMv4T, the Thumb instruction set is supported. It is 16-bit and aims to improve compiled code density [19]. Since ARMv6T2, additional 32-bit instructions are also introduced to extend the Thumb instruction set (T32). ARMv8a provides an optional 64-bit architecture named "AArch64", and also an associated new ARM instruction set (A64) to provide the access to 64-bit general-purpose registers. At the same time, it still maintains compatibility with 32-bit architectures and inherits A32 and T32. When an application is executed on an ARMv8 processor, it could be in either the AArch32 state (using A32 or T32 instructions) or the AArch64 state (using A64 instructions). In total, there are three instruction sets used by ARM binaries, i.e., A32, T32, and A64. For discussion simplicity, we will focus on the 32-bit ARM architecture, which uses A32 and T32 instruction sets, while our system also supports A64.

Unlike x86/x64 that uses a single instruction set, a 32-bit ARM binary usually contains both A32 and T32 instructions. Note that T32 contains both 16-bit and 32-bit instructions. As the A32 and T32 instructions share the same encoding space, a sequence of four bytes could be decoded as either an A32 instruction or as one or two T32 instructions. This makes ARM disassembly challenging.

B. ARM/Thumb Interworking

The A32 and T32 instructions provide almost the same functionality. An ARM binary usually uses both instruction sets together to achieve both high performance and better code density. Although the A32 and T32 instructions do not overlap, they may interleave. An ARM processor in operation can be in the ARM mode, executing the A32 instructions, or in the Thumb mode, executing the T32 instructions. The mode is determined by the `T` bit in the *Current Program State Register (CPSR)*. When `T` is 0, the processor gets into the ARM mode, 1 the Thumb mode. The `T` bit can be set by the least significant bit of a branch target in a branch instruction, e.g., by the branch with link and exchange instruction (`blx`) and the branch and exchange instruction (`bx`). Switching between the two modes can be achieved explicitly or implicitly. For example, the instruction `blx label` always changes the current mode, while `blx rm` and `bx rm` set the `T` bit as the least significant bit of the branch target in the register `rm` and may or may not trigger the mode change. Some other instructions such as `pop`, `ldr`/`ldm` and some arithmetic instructions may also change the instruction mode implicitly, when writing into the program counter register `pc` and causing a control transfer. With these implicit mode switches, the instruction mode could only be determined at runtime, which makes static disassembly very challenging.

C. Inlined Data

Inlined data may cause a lot of false positives and false negatives for disassembly. In x86/x64, it is believed that inlined data is not prevalent [20] and, hence, may not be that problematic in practice. However, this is not true in ARM. As described above, the instructions in ARM binaries

are only 16 or 32 bits long. Constants in instructions, e.g., operands that are immediate, must be encoded as part of the 16 or 32 bits, which limit the range of constants that can be used in a single instruction. When a constant cannot be encoded as an immediate operand, ARM usually loads it from memory and moves to a register as an operand. Thus, data are commonly inlined in code sections and load/store instructions are frequently used in ARM. In Section VI-A, we show that on average 4.7% of an ARM binary is inlined data, substantially more than an x86 binary, which usually has only 0-1% of inlined data [20]. Note that, although 64-bit ARM binaries do not have interleavings, AArch64 is still a RISC architecture and introduces lots of inlined data.

III. Motivation

In this section, we use an example to show the limitations of existing disassembly techniques and motivate ours.

A. Motivating Example

Figure 1(a) presents a code snippet from *bzip2* in SPEC CPU2000, compiled with `GCC -marm -O1 -march=armv7-a` but slightly modified for the illustration purpose. Its functionality is irrelevant. As mentioned in Section II, A32 and T32 instructions are either 2-byte or 4-byte aligned. We list the 2-byte aligned virtual addresses and the corresponding raw bytes in the hex form in the first and the second columns, respectively. In the third and fourth columns, we show all the instructions that could be legally decoded starting from each address, respectively. Observe that except the four bytes starting at `0x2dc20` that could not be decoded as A32 instructions, all the other 4-byte sequences could be decoded by both instruction sets.

The ground truth of the snippet, that is, the true instruction sequences, is highlighted by the green boxes. It consists of two code sections, one in ARM and the other in Thumb, and an interleaved data section (in the second column starting at `0x2dc20`). The code starts in the ARM mode. In ARM, individual A32 instructions can be used for conditional execution to save code space whilst improving performance. An instruction with a conditional code suffix, e.g., the `eq` suffix, reads the corresponding flag in the *CPSR* register, e.g., the equivalence flag, to determine whether or not to be executed. For example, the first `cmp` instruction (at address `0x2dbf8`) compares the value of register `r6` and zero and sets the corresponding conditional flag(s) in the *CPSR* register. The `pl` suffix of the next instruction `addpl` at address `0x2dbfc` means "positive or zero". It is executed only if the `N` (Negative) flag in *CPSR* is disabled, i.e., `r6` $\geq$ 0 in the first `cmp` instruction. The `eq` suffix in the following instructions means equivalence in a similar strain of denotation.

Within the conditional code zone (in the ARM box), the instruction `add r5, r7, r8, lsl #1` computes the resultant value of `r7 + r8 << 1` and stores it into `r5`. The `movw` and `movt` instructions at addresses `0x2dc00` and `0x2dc04` set the lower and higher 16 bits of `r6` separately to make it `0x2dc25`. The subsequent two instructions first set `r1` as the value pointed to by `r5` via `mov r1, r5`, and then store `r6` into the target address in `r1` via `str r6, [r1]`. Now the bytes at the memory address denoted by `[r1]` (also `[r5]`) is `0x2dc25`. Next, `sub r6, #5` updates the value of `r6` to `0x2dc20` by subtracting it by `5`, and `ldr r6, [r6]` reads the 4-byte value located at `r6`, i.e., `0x2dc20`, which is actually the starting address of inlined data. At the end of the ARM section, `ldr r0, [r5]` sets the value of `r0` as the bytes at `[r5]`, which is `0x2dc25` according to the `str` instruction at address `0x2dc0c`. The target destination of the instruction `bx r0` is hence `0x2dc25`, whose least significant bit is 1. As mentioned in Section II-B, it branches to address `0x2dc24` and also switches to the Thumb mode.

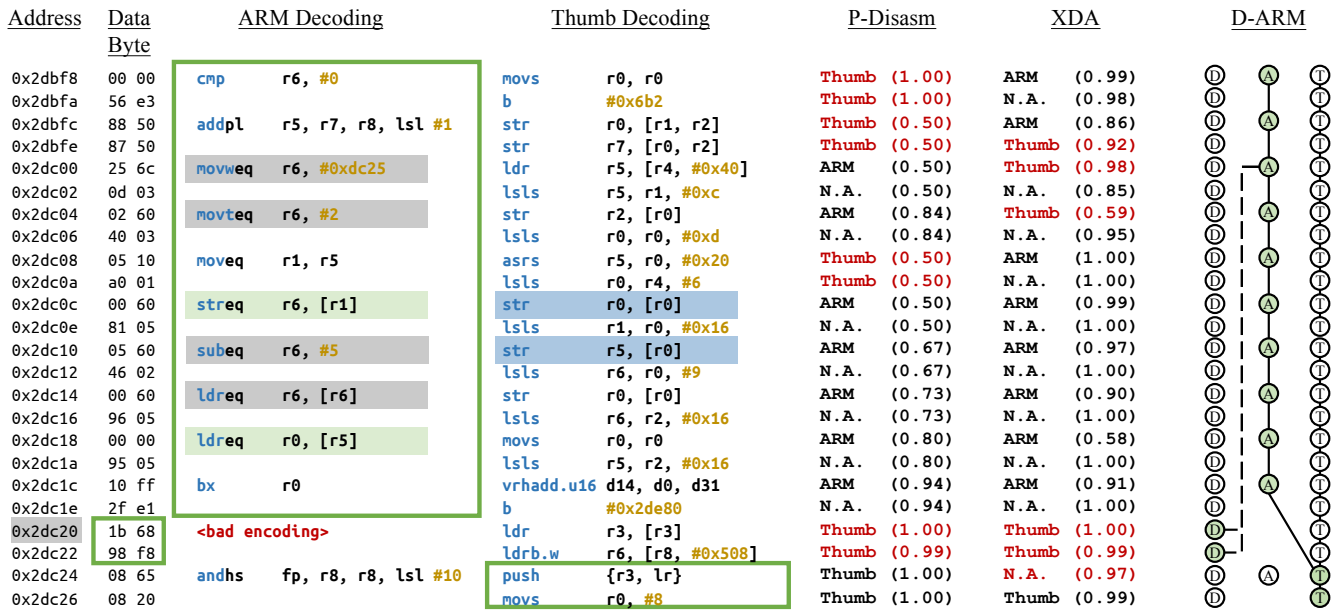
B. Limitation of Existing Techniques

A variety of strategies and algorithms have been proposed for binary disassembly. However, almost all of them [15], [13], [14] are designed for x86/x64 binaries and could not work well for ARM binaries in consideration of the features we discussed in Section II. In this subsection, we show that the interleaving of the two modes and the inlined data in the example in Figure 1 (a) pose great challenges to existing disassembly techniques.

Linear Sweep Disassembly. Linear sweep disassemblers, such as Objdump, simply scan code sections and disassemble instructions following the address order. However, it cannot tell which instruction set to use even given the correct start address of a code section, not to mention the complex instruction set interleaving within instruction sequences. Also, inlined data cannot be detected, incurring a lot of false positives (i.e., data bytes are recognized as instructions). In our example, a linear sweep disassembler may decode all bytes as T32 instructions.

Recursive Traversal Disassembly. Recursive traversal disassembly starts from function entries and disassembles instructions following control flow edges. Many popular disassemblers, such as IDA [11] and Ghidra [12], are based on this strategy to reduce false positives. However, the major disadvantage of this strategy is that code blocks may be missed if they are reached through indirect jumps or calls. In ARM, this problem becomes more prominent due to the limited immediate target range that could be used for direct branching, as we discussed in Section II-C. For example, in Figure 1 (a), the recursive traversal disassembly may miss the Thumb instructions starting at address `0x2dc24`, as it is reached by an indirect branch `bx r0` at `0x2dc1c`.

Probabilistic Disassembly. Probabilistic disassembly (P-Disasm) [14] is a recent approach for binary disassembly and rewriting. It generates a superset of instructions [21] by considering each address in the code space as the start address of some instructions, and then computes a probability for each address to indicate its likelihood of being a true positive instruction. Probabilities are computed from a set of hints, including control flow convergence, control flow crossing, and register definition-use relation. Its experiments show that it has no false negatives and only 3.7% false positives on x86 binaries.



(a) Example Code Snippet (b) Baseline Tools' Results (c) D-ARM

Fig. 1: Motivating example

In the superset instructions, there exist a lot of occluded instructions that overlap with each other. P-Disasm is based on an assumption that occluded instruction sequences tend to quickly converge on true instructions (usually within four instructions). Intuitively, it means that if the tool starts to disassemble at the wrong address, it can quickly correct itself and find a true starting address. This is a unique property of x86 instruction set design. However, this is not true in ARM binaries. As mentioned in Section II, both ARM and Thumb instructions have fixed instruction lengths (2-byte aligned or 4-byte aligned). If initially the mode is wrong, such a mistake can carry on forever, without any self-correction ability. To make the situation worse, the incorrectly disassembled instruction sequences also have substantial control flow and data flow hints that make them look legitimate. For example in Figure 1(a), although the instructions from `0x2dbf8` to `0x2dc1f` (20 bytes in total) are in the ARM mode, if one disassembled them in Thumb, the resulting instructions look legitimate. For example, there is a definition-use relation between the instructions at `0x2dc0a` and `0x2dc0c`, and between `0x2dc0e` and `0x2dc10`. Since these are hints used by P-Disasm to decide validity, they cause substantial confusion for the tool. We call the situation *instruction set occlusion*, which exists for the entire code sections.

In Figure 1(b), the left column shows the outcomes of P-Disasm, presented in the form of "`type (prob)`" where `prob` is the computed posterior probability for the instruction set type of that address, with `N.A.` for non-instruction. The false positives are highlighted in red. We can see that, in the Thumb instruction sequence, the general register `r0` and `r5` happen to be defined and used for many times, which makes many Thumb instructions (`0x2dbf8` to `0x2dbfe` and `0x2dc08` to `0x2dc0a`) achieve high probabilities and causes a lot of false positives. The lengthy occluded sequences also make it more difficult for the probability computation to reach a fixed point. In Section VI, our experiments show that the ARM version of P-Disasm terminates for many binaries due to memory explosion.

Another type of errors is caused by inlined data. The false instructions decoded from the inlined data could also have many hints, supporting their legitimacy. This can lead to false positives. In this case, the data section is decoded as two Thumb instructions (`0x2dc20` and `0x2dc22`) by P-Disasm.

XDA. Besides the traditional rule-based algorithms, some machine learning (ML) models have also been explored for disassembly. XDA [13] is a recently proposed disassembly framework based on transfer learning. It takes the raw bytes as input and uses masked language modeling to learn machine code dependencies. Then the pretrained model is further finetuned for different downstream tasks, such as disassembly and function boundaries recovery.

However, as a common issue of ML-based methods, it is hard to interpret the models, which more often use syntactic patterns instead of semantics when making predictions. For example, the right column of Figure 1(b) shows the results of XDA. Although the first two ARM instructions (`0x2dbf8` and `0x2dbfc`) are correctly identified, the Thumb instructions at the following addresses (`0x2dbfe` to `0x2dc04`) are falsely given higher probabilities. This is possibly because the `str` and `ldr` Thumb instructions are very common and learned by the model. However, these errors could be avoided if semantics were considered, as the ARM instruction `add r5, r7, r8, lsl #1` at `0x2dbfc` does not change the mode and the following instructions should stay in ARM. Even if XDA learned some semantics, such information would be very local. This is because XDA splits all bytes into fixed

length sequences (e.g., 512 bytes) as input and decodes these sequences separately, which means the instructions along some non-trivial control flow path may be cut into sequences and the cross-sequence semantics are lost. Similar to P-Disasm, XDA also decodes the 4 data bytes at `0x2dc20` as Thumb instructions. When inlined data happens to share syntactic forms with common instructions, e.g., the `ldr` instruction at `0x2dc20`, they are likely classified as instructions by XDA.

C. Our Technique

Insights. Our tool is inspired by two important insights. The first insight is that rich semantic constraints can be leveraged. Specifically, the true instructions should satisfy certain constraints, e.g., two true instructions should not overlap with each other, and instructions should or should not be decoded at the same time. For example in Figure 1 (a), if the ARM instruction `cmp r6,#0` at `0x2dbf8` is true, the two Thumb instructions `movs r0, r0` and `b #0x6b2` at the same addresses must be false as they overlap with the ARM instruction. Also, at the address `0x2dbfc`, the ARM `add` instruction should be decoded and the two Thumb `str` instructions should not, because the prior `cmp` instruction does not change the mode and it should hence still be in the ARM mode. Similarly, all the following ARM instructions will be decoded until it reaches the branch instruction `bx r0` at `0x2dc1c`. If we know the value stored in `r0` is `0x2dc25` (as explained in Section III-A), the Thumb instruction `push r3, lr` at `0x2dc24` should be decoded as the mode changes. There are also constraints between instructions and inlined data. For example, if the ARM instruction `ldr r6, [r6]` at `0x2dc14` is true, the bytes at `0x2dc20` must be data when the memory address denoted by `r6` can be determined as `0x2dc20` (explained in Section III-A), and the Thumb instruction at this address hence false.

The second insight is that interpreting complex program behaviors can help find true instructions. As a RISC architecture, ARM tends to use memory to store temporary variables. Such memory behaviors usually occur within a local context (i.e., a consecutive instruction sequence without loop or recursion) and can provide a wealth of information regarding indirect branching, instruction mode switching, and data regions. For example, if we can correctly interpret the ARM instructions as the way we discuss in Section III-A, the memory address for the `ldr` instruction at `0x2dc14` and the `bx` instruction at `0x2dc1c` can be determined, and also the corresponding constraints mentioned above. In addition, lengthy occluded sequences imply a large number of bogus hints, such as the register definition and use hints used by P-Disasm. In contrast, interpreting memory behaviors allows finding false instruction sequences. For example, in the false Thumb instruction sequences in Figure 1 (a), the `str` instruction at `0x2dc10` overrides the same memory region after the `str` at `0x2dc0c` (highlighted in blue) without any reading operation. This meaningless behavior degrades the probability of the two Thumb instructions and even other instructions in the sequence being true.

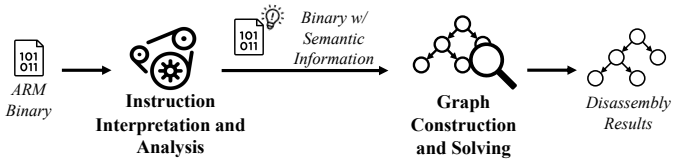


Fig. 2: Architecture of D-ARM

Our Idea. Inspired by the above insights, we devise a two-staged disassembly technique for ARM binaries. The architecture of D-ARM is shown in Figure 2. In the first stage, a lightweight static analysis is used to interpret all superset instructions and collect rich semantic information such as memory accesses, indirect control flow, and dependencies. In the second stage, we model the binary as a graph. For each address, we define three nodes to denote that it could be an ARM instruction, a Thumb instruction, or data. For our example, as shown in Figure 1 (c), each address (e.g., `0x2dbf8`) has three nodes, namely, Ⓐ denoting the address being an ARM instruction, Ⓓ data bytes, and Ⓣ a Thumb instruction. A weight value is derived for each node by counting the number of semantic relations in which the node is involved. Edges are further introduced to denote the semantic correlations across nodes derived by the instruction interpretation. For example the edge between the Ⓐ nodes at addresses `0x2dbf8` and `0x2dbfc` denotes the control flow dictated by the instruction order. The edge between Ⓐ@`0x2dc00` and Ⓓ@`0x2dc20/0x2dc22` denotes a memory access and that between Ⓐ@`0x2dc1c` and Ⓣ@`0x2dc24` denotes an indirect control transfer. Here, only a subset of the edges are shown for simplicity. Details of graph construction and examples can be found in Section V. The disassembly problem is then reduced to the problem of finding a *maximum weight independent set* (MWIS) [17] in the graph. The MWIS contains the richest semantic information and respects all constraints denoted by the graph edges. It hence represents the disassembly results. The graph problem can be solved by a greedy algorithm. In our example, D-ARM successfully identifies the two code regions and the inlined data.

IV. Superset Instruction Interpretation

The goal of the static analysis is to have a low-cost method to model the behaviors of the target binary, especially its memory behaviors. That is, a load from an address should return the value defined by the closest store. As such, the values at each instruction can be derived accurately, which is critical for extracting higher-order semantic properties, such as implicit instruction mode switches, memory dependencies, and indirect control transfer targets. However, since addresses are statically unknown, memory behavior modeling is very difficult by conventional static analysis, e.g., a sound abstract interpretation would produce over-approximation that leads to a lot of bogus information, degrading our analysis. Our overarching idea is to have lightweight modeling that facilitates derivation of program properties important for correct disassemly over all superset instructions. Our analysis results are neither over-approximation nor under-approximation. In

particular, it disregards if an instruction is true, which is unknown. Instead, it pretends it is true and interprets it anyway. We hence call it *superset interpretation*. It captures part of memory behaviors at a low cost. We then rely on the graph analysis stage to aggregate such partial information across the whole program. The aggregation substantially suppresses errors in the analysis stage and allows the true disassembly results to stand out. Intuitively, one can think of the second stage as a massive voting step driven by the graph structure inherent in the program, each superset instruction having its own vote and all votes being aggregated following the graph structure. Note that since the analysis is on all superset instructions, including the false instructions, it produces a certain amount of bogus information, i.e., infeasible program behaviors.

The static analysis is driven by a lightweight instruction interpretation technique on an abstract domain. We use an affine expression over a set of symbolic values to denote the abstract values produced by the interpretation of a superset instruction. When D-ARM cannot decide the abstract value at an instruction, e.g., when the instruction loads an external value or loads from an address whose affine expression does not match the expression of any preceding store (for example, because it is not a true instruction), a symbolic value is introduced to denote the value at that instruction. Any following computation using this value leads to an affine expression with the symbolic value. Two values are considered identical only when their affine expressions are the same. This is somehow incomplete, that is, equivalent values may have different affine expressions. However, as discussed earlier, this is reasonable in our context as errors can be tolerated in the graph analysis stage.

We derive rich semantics from the affine expressions. For an indirect control transfer, we reduce/concretize an expression of the target register to a set of constant values denoting the possible targets. By identifying loads and stores that have the same address value (and no intermediate stores to the same address), we extract memory dependencies. However, in most cases, the affine expressions do not directly provide instruction mode information because the least significant bit can hardly be statically determined if any symbolic value is involved (except in some special cases, e.g., when it can be determined that the values denoted by the symbolic expression are dividable by 2). We hence leverage the control flow and memory dependencies information in the later graph analysis phase to infer instruction mode.

A. Language and Abstract Domain

Abstract Language. To facilitate discussion, we introduce a low-level but abstract language to model ARM binaries. The language is designed to illustrate our key ideas and, hence, omits many irrelevant features of ARM. The implementation of D-ARM, on the other hand, fully supports disassembling real-world ARM binaries. The syntax of the language is presented in Figure 3. A binary is denoted by a mapping from $\langle \mathit{Address}, \mathit{InstrSet} \rangle$ to an instruction where D-ARM uses a non-negative integer $\mathit{Address}$ to denote the virtual address and a Boolean variable $\mathit{InstrSet}$ to distinguish the different instruction modes of ARM. $\mathbf{R}(r_a)$ and $\mathbf{W}(r_a, r_v)$ model the memory read and write operations, respectively, where register r_a holds the memory address and r_v holds the value to write. A direct control transfer is modeled by $\mathbf{goto}(a, i)$. The transfer target address is denoted by a and the instruction mode after transfer is explicitly denoted by i. The guarded **goto**, represented as **if** r_c **goto(a, i)**, models a conditional statement in which the predicate outcome in register r_c dictates the transfer. Handling indirect control flow is particularly challenging for disassembly (e.g., `bx r0` in Figure 1(a)). D-ARM uses $\mathbf{i\text{-}goto}(r_t)$ and **if** r_c $\mathbf{i\text{-}goto}(r_t)$ to model indirect jumps and conditional indirect jumps, respectively. Register r_t holds the target address of **i-goto** and the CPU determines the new instruction mode by examining the last bit of r_t.

$\langle \mathit{Binary} \rangle$	B	::=	$\langle a,\ i \rangle \rightarrow I$
$\langle \mathit{Instruction} \rangle$	I	::=	$r := E$ | $r := \mathbf{R}(r_a)$ | $\mathbf{W}(r_a,\ r_v)$
		|	$\mathbf{goto}(a,\ i)$ | **if** r_c $\mathbf{goto}(a,\ i)$
		|	$\mathbf{i\text{-}goto}(r_t)$ | **if** r_c $\mathbf{i\text{-}goto}(r_t)$
$\langle \mathit{Expression} \rangle$	E	::=	r | c | $r_1\ op\ r_2$ | $r\ op\ c$
$\langle \mathit{Register} \rangle$	r	::=	$\{\mathbf{sp},\ \mathbf{r}_0,\ \mathbf{r}_1,\ \mathbf{r}_2,\ \ldots\}$
$\langle \mathit{Operator} \rangle$	op	::=	$+$ | $-$ | $*$ | $\div$ | BitOp | ...
$\langle \mathit{Const} \rangle$	c	::=	$\mathbb{Z}_{\geq 0}$
$\langle \mathit{Address} \rangle$	a	::=	$\mathbb{Z}_{\geq 0}$
$\langle \mathit{InstrSet} \rangle$	i	::=	0 (**ARM**) | 1 (**Thumb**)

Fig. 3: A simple language for pre-disassembling binaries

Abstract Domain. We describe D-ARM's abstract domain in the following. We introduce a symbolic value type called *address descriptor (AD)*.

Definition IV.1 (AD)**.** *An address descriptor $AD^{\langle a,\ i \rangle}$, where $a \in$ Address is a valid address in the virtual space of the given binary B, and $i \in$ InstrSet is a constant instruction mode, denotes the value computed by the instruction $I \equiv B\,[a,\ i]$.*

Intuitively, $AD^{\langle a,i \rangle}$ is introduced when the abstract value of instruction $B\,[a,\ i]$ cannot be determined statically. Such symbolic value can be used in the interpretation of following instructions that rely on the result of this instruction.

Example. Table I presents a crafted example for the illustration purpose. Above the table is a source code snippet taking an integer array `p` and an index `i` as parameters and returning the element `p[i]`. The three columns present the assembly code decoded as A32 instructions, D-ARM's low-level language representation, and the evaluation results during the interpretation, respectively. Each A32 instruction is preceded by its address. Specifically, the address of `p` and the value of `i` are held by `r0` and `r1`, respectively. The first instruction loads the value of `p` into `r2` while the second one loads `i` into `r3`. The third and fourth instructions compute the offset between `p[i]` and `p` and store it into `r4`. In particular, `r4` is first set as 4, the size of an integer, and then multiplied by `r3` (i.e., `i`). By adding `r2` to the computed offset at address `10`, `r4` is set to the memory address of `p[i]`, namely `&(p[i])`. The last instruction performs pointer dereference of `r4` to get the value of `p[i]`.

Observe that the value held by register `r0` at address `00` cannot be determined statically because it is from the external

TABLE I: Example of abstract values

Source code:
```
int f(int p[], int i){ return p[i]; }
```

Assembly Code (ARM)	Trace	Evaluation Results
00. **mov** r2, r0	$r2 := r0$	$\text{AD}^{\langle 0,0\rangle}$
04. **mov** r3, r1	$r3 := r1$	$\text{AD}^{\langle 4,0\rangle}$
08. **mov** r4, #4	$r4 := 4$	4
0c. **mul** r4, r3, r4	$r4 := r3 \times r4$	$4 \times \text{AD}^{\langle 4,0\rangle}$
10. **add** r4, r4, r2	$r4 := r4 + r2$	$\text{AD}^{\langle 0,0\rangle} + 4 \times \text{AD}^{\langle 4,0\rangle}$
14. **ldr** r0, [r4]	$r0 := \mathbf{R}(r4)$	$\text{AD}^{\langle 14,0\rangle}$

$pc \in \texttt{ProgramCounter} ::= \texttt{Address} \times \texttt{InstrSet}$
$EI \in \texttt{ExploredInstr} ::= \{\, \texttt{Address} \times \texttt{InstrSet} \,\}$
$RS \in \texttt{RegisterStore} ::= \texttt{Register} \rightarrow \texttt{AbstractValue}$
$MS \in \texttt{MemoryStore} ::= \texttt{AbstractValue} \rightarrow \texttt{AbstractValue}$
$PS \in \texttt{ProgramState} ::= \langle \texttt{Address} \times \texttt{InstrSet} \rangle \rightarrow \langle \texttt{RegisterState} \times \texttt{MemoryState} \rangle$

Fig. 4: Definitions

parameter p. D-ARM uses $AD^{\langle 0,0\rangle}$ to represent the evaluation outcome, where the first 0 denotes the address and the second 0 determines the instruction set is ARM. $AD^{\langle 4,0\rangle}$ at address 04 represents i in a similar strain of denotation. □

Linear operations are faithfully interpreted such that the abstract value for each (superset) instruction must be an affine expression over a set of symbolic values. For operations that we do not model, such as multiplications of operands with non-constant affine expressions, new symbolic values are introduced. Note that this is sufficient to model memory behaviors as address computations are linear. For instance, offsets can be represented as affine expressions of index variables.

Definition IV.2 (V)**.** *An abstract value V is an affine expression over address descriptors, denoted by $V \equiv c_0 + \sum_k c_k \times AD^{\langle a_k,\, i_k\rangle}$.*

Example Continued. In Table I, "08. **mov** r4, #4" assigns a constant value to r4. Hence its abstract value is 4. At address 0c, r3 is multiplied by r4, where r3 holds an unknown external input i with an abstract value $AD^{\langle 4,0\rangle}$. The evaluation result is hence $4 \times AD^{\langle 4,0\rangle}$. Another unknown input p is added to r4 at address 10. Hence the abstract value is $AD^{\langle 0,0\rangle} + 4 \times AD^{\langle 4,0\rangle}$. The last pointer dereference at address 14 yields a new address descriptor $AD^{\langle 14,0\rangle}$ as there is no preceding store to the address $AD^{\langle 0,0\rangle} + 4 \times AD^{\langle 4,0\rangle}$. □

B. Overall Procedure

Given a binary, the analysis selects a (superset) instruction with the lowest address value that has not been interpreted and performs interpretation. It then follows the control flow behavior of the instruction to interpret the next until it reaches an instruction that has been interpreted before. It then repeats the process until all superset instructions have been interpreted. This implies D-ARM interprets each loop body only once. If the branch outcome of a conditional instruction cannot be determined, D-ARM randomly chooses one, otherwise it follows the program semantics. The interpretation process models both register and memory reads and writes, e.g., supporting writing an abstract value to an abstract address. Recall that ARM, as a RISC architecture, tends to use memory to store temporary variables, which often takes place within a local context (e.g., within a basic block or nearby basic blocks). The loop- and recursion-free strategy allows the process to be lightweight. The overall algorithm is elided due to its simplicity. Note that our technique does not require a perfect control flow graph. Instead, it just aims to interpret all instructions, including those incorrectly disassembled, such that semantic information can be collected for the later graph analysis. In particular, the static analysis tries to traverse along a control flow path as far as possible and concretizing indirect jump targets is just one of such efforts. It does not hurt if jump targets are missing as the analysis will select an uncovered address for the next round of interpretation.

$$V_x + V_y = (c_0^x + c_0^y) + \left(\sum_k c_j^x \times AD^{\langle a_j^x,\, i_j^x\rangle} + \sum_k c_k^y \times AD^{\langle a_k^y,\, i_k^y\rangle} \right)$$

$$V_x - V_y = (c_0^x - c_0^y) + \left(\sum_j c_j^x \times AD^{\langle a_j^x,\, i_j^x\rangle} - \sum_k c_k^y \times AD^{\langle a_k^y,\, i_k^y\rangle} \right)$$

$$V_x \times V_y = \begin{cases} c_0^x \times c_0^y + \sum_j c_j^x \times c_0^y \times AD^{\langle a_j^x,\, i_j^x\rangle} & \texttt{if } V_y = c_0^y \\ c_0^x \times c_0^y + \sum_k c_k^y \times c_0^x \times AD^{\langle a_k^y,\, i_k^y\rangle} & \texttt{if } V_x = c_0^x \\ \top & \texttt{otherwise} \end{cases}$$

$$V_x \div V_y = \begin{cases} c_0^x \div c_0^y + \sum_j c_j^x \div c_0^y \times AD^{\langle a_j^x,\, i_j^x\rangle} & \\ \qquad \texttt{if } V_y = c_0^y \wedge c_0^y \mid \gcd(c_0^x, c_1^x, \ldots) & \\ \top & \texttt{otherwise} \end{cases}$$

$\texttt{BitOp}(V_x, V_y) = \top$

Fig. 5: Arithmetic operations over affine expressions. Without loss of generality, we have $V_x = c_0^x + \sum_k c_j^x \times AD^{\langle a_j^x,\, i_j^x\rangle}$ and $V_y = c_0^y + \sum_k c_k^y \times AD^{\langle a_k^y,\, i_k^y\rangle}$

C. Abstract Semantics

In this section, we discuss the semantics of instruction interpretation, that is, how individual instructions are interpreted. Figure 4 introduces a number of definitions that are used in the semantic rules. We use *pc*, namely program counter, to denote the location of a superset instruction. In our context, *pc* contains an additional constant instruction mode InstrSet, to distinguish the different instructions (in different modes) at the same address. *EI* denotes the set of interpreted superset instructions. *RS* denotes the register store that maps a register to its abstract value, and *MS* denotes the abstract memory store that maps an abstract memory address value to an abstract value (stored at that address). Program state, denoted by *PS*, includes both the register and memory stores.

Figure 5 presents the rules for arithmetic operations over abstract values (i.e., affine expressions), where we assume two operands V_x and V_y. Addition and subtraction operations are interpreted as the addition and subtraction of the affine expressions of the operands. It is easy to infer that the resulting abstract value is still affine. Multiplication can only be interpreted if one of the operands is a constant c_0. The result is the abstract value of the other operand multiplied by c_0. Otherwise, the multiplication yields an unknown value $\top$

TABLE II: Interpretation rules

Rule	Statement	Actions
READ	$r \coloneqq \mathbf{R}(r_a)$	$EI \coloneqq \{pc\} \cup EI$; $\langle RS, MS\rangle \coloneqq PS[pc]$; $v \coloneqq RS[r_a]$; $RS[r] \coloneqq (v \equiv \bot \vee MS[v] \equiv \bot \ ?\ AD^{pc}\ :\ MS[v])$; $pc \coloneqq pc.\texttt{next}()$; $PS[pc] \coloneqq \langle RS, MS\rangle$;
WRITE	$\mathbf{W}(r_a, r_v)$	$EI \coloneqq \{pc\} \cup EI$; $\langle RS, MS\rangle \coloneqq PS[pc]$; $v \coloneqq RS[r_a]$; `if` $(v \not\equiv \bot)$ $\{MS[v] \coloneqq (RS[r_v] \equiv \bot\ ?\ AD^{pc}\ :\ RS[r_v])\}$; $pc \coloneqq pc.\texttt{next}()$; $PS[pc] \coloneqq \langle RS, MS\rangle$;
EXPR	$r \coloneqq r_1\ op\ r_2$	$EI \coloneqq \{pc\} \cup EI$; $\langle RS, MS\rangle \coloneqq PS[pc]$; `if` $(RS[r_1] \equiv \bot \vee RS[r_2] \equiv \bot \vee RS[r_1]\ op\ RS[r_2] \equiv \top)$ $\{RS[r] \coloneqq AD^{pc}\}$ `else` $\{RS[r] \coloneqq RS[r_1]\ op\ RS[r_2]\}$; $pc \coloneqq pc.\texttt{next}()$; $PS[pc] \coloneqq \langle RS, MS\rangle$;
GOTO	**goto**(a, i)	$EI \coloneqq \{pc\} \cup EI$; `if` $(B[\langle a, i\rangle] \not\equiv \bot)$ $\{PS[\langle a, i\rangle] \coloneqq PS[pc]$; $pc \coloneqq \langle a, i\rangle;\}$ `else` $\{pc \coloneqq nil;\}$;
I-GOTO	**i-goto**(r_t)	$EI \coloneqq \{pc\} \cup EI$; $\langle RS, MS\rangle \coloneqq PS[pc]$; `if` $(RS[r_t] \equiv c_0)$ $\{$**goto**$(c_0 - c_0 \bmod 2, c_0 \bmod 2);\}$ `else` $\{pc \coloneqq nil;\}$;
IF-GOTO	**if** r_t **goto**(a, i)	$EI \coloneqq \{pc\} \cup EI$; $\langle RS, MS\rangle \coloneqq PS[pc]$; `if` $(B[pc.\texttt{next}()] \not\equiv \bot)$ $\{PS[pc.\texttt{next}()] \coloneqq \langle RS, MS\rangle;\}$; `if` $(B[\langle a, i\rangle] \not\equiv \bot)$ $\{PS[\langle a, i\rangle] \coloneqq \langle RS, MS\rangle;\}$; `if` $(RS[r_c] \not\equiv c_0)$ $\{pc \coloneqq (\texttt{rand}()\ ?\ pc.\texttt{next}()\ :\ \langle a, i\rangle)\}$ `else` $\{pc \coloneqq (RS[r_c] \equiv 0\ ?\ pc.\texttt{next}()\ :\ \langle a, i\rangle)\}$;
IF-I-GOTO	**if** r_c **i-goto**(r_t)	$EI \coloneqq \{pc\} \cup EI$; $\langle RS, MS\rangle \coloneqq PS[pc]$; `if` $(B[pc.\texttt{next}()] \not\equiv \bot)$ $\{PS[pc.\texttt{next}()] \coloneqq \langle RS, MS\rangle;\}$; `if` $(RS[r_c] \not\equiv c_0)$ $\{pc \coloneqq nil\}$; `if` $(RS[r_c] = 0)$ $\{pc \coloneqq pc.\texttt{next}()\}$ `else` $\{$**i-goto**$(r_t)\}$;

which will be replaced by a new address descriptor during interpretation. Observe the result is still an affine expression. Division is similarly interpreted only when the divisor is a constant and the coefficients of the divident are all divisible by the constant. Other operations are not interpreted and the resulting value is $\top$.

Semantic Rules. The semantic rules are presented in Table II. Upon interpreting an instruction, the current *pc* is marked as interpreted, then *RS* and *MS* at the current program point are updated based on the semantics of the instruction. After the interpretation of an instruction, *pc* is first set as the location of the next instruction (with the same instruction mode) and the current *RS* and *MS* are then propagated to the new *pc*, unless otherwise stated. Note that although D-ARM interprets instructions in an abstract domain, it is different from conventional abstract interpretation [22], which performs join operations and usually weak updates due to its soundness requirement. In contrast, D-ARM is unsound and even interprets bogus instructions. The goal is to derive sufficient semantic information for correct disassembly. Rule **READ** describes the semantics of memory read. It first queries the abstract address value v in register r_a, denoted as $RS[r_a]$. If the value of register r_a or the memory at the abstract address v is not defined, a symbolic value AD^{pc} is assigned to the target register r, otherwise $MS[v]$, the content at address v. Rule **WRITE** describes the semantics of memory write. Similar to memory read, it first examines the value v in register r_a. If v is valid, the memory $MS[v]$ is updated as $RS[r_v]$ or AD^{pc} depending on whether register r_v is defined or not. Note that when v is not defined, a conventional abstract interpretation technique [23] becomes remarkably onerous by updating the memory content for all possible addresses, introducing a large number of bogus memory behaviors and rendering the interpretation results largely useless. The rationale is that traditional analysis has to be conservative for its applications such as compiler optimizations. However, in our context, we prefer to suppress bogus behaviors, even with the cost of missing some behaviors. Rule **EXPR** evaluates an assignment with expression evaluation. If either register r_1 or r_2 is not defined, or the evaluation results cannot be represented by an affine expression over address descriptors, AD^{pc} is assigned to register $RS[r]$. Otherwise, it is set as the evaluation result of the operation, following the rules in Figure 5. Rule **GOTO** sets the program counter to the target location $\langle a, i\rangle$. Note that the current *MS* and *RS* are propagated to the target location instead of `pc.next()`. Rule **I-GOTO** describes the semantics of indirect jumps. If the abstract value of register r_t does not contain any symbolic value, i.e., it is a constant c_0, D-ARM derives a **goto**$(c_0 - c_0 \bmod 2, c_0 \bmod 2)$ statement (i.e., by simulating the behaviors of ARM CPU) and then interprets it. Otherwise, D-ARM sets *pc* as *nil* and terminates this round of interpretation. Note that even though D-ARM terminates interpretation when the control transfer target is not constant, the instructions at the target address will nonetheless be interpreted, driven by the overall algorithm, which interprets all superset instructions. In Rule **IF-GOTO**, D-ARM validates the two outgoing branches and propagates the current *RS* and *MS* to the valid one. If $RS[r_c]$ is not a constant, indicating that D-ARM cannot determine the branch outcome statically, a random branch is chosen. D-ARM resorts to the program semantics otherwise. Similar to Rule **IF-GOTO**, Rule **IF-I-GOTO** first examines the predicate $RS[r_c]$ and derives an **i-goto**(r_t) for further interpretation. A running example of the analysis is presented in Appendix A.

V. GRAPH ANALYSIS

In the second stage, the binary is modeled as a graph. The semantic information derived from the previous stage is denoted as edges and node weights. The disassembly problem is hence reduced to a *maximum weight independent set* (MWIS) problem in graph theory [17], aiming at finding an optimal sub-graph that can satisfy a set of given constraints and have the largest aggregated weight, i.e., expressing maximum semantic information. The sub-graph denotes the disassembly results. In the following, we first define the graph and then explain how we solve it.

A. Nodes

For each address a, we introduce a node $\langle a, 0\rangle$ to denote that it is an ARM instruction, a node $\langle a, 1\rangle$ to denote that it is a Thumb instruction, and a node $\langle a, -1\rangle$ to denote a is not an instruction, but rather inlined data. We denote all the ARM (superset) instructions as N_A, all the Thumb (superset) instructions as N_T, and all the data nodes as N_D.

The node set N of the graph is hence the following.

Definition V.1 (N). $N = N_A \cup N_T \cup N_D$.

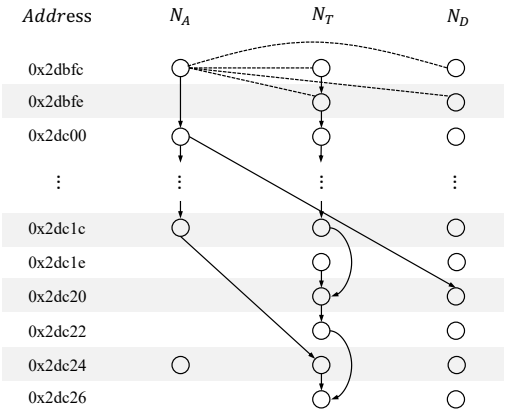


Fig. 6: Graph for the example in Figure 1(a)

Example. Figure 6 shows the generated graph model for the bytes from address `0x2dbfc` to `0x2dc26` in the motivation example in Figure 1(a). Only part of the edges are displayed in Figure 6 for explanation simplicity. The ARM node $\langle$`0x2dbfc`$, 0\rangle$, i.e., the node at the `0x2dbfc` row and N_A column, represents the ARM `add` instruction at `0x2dbfc` in Figure 1(a). The Thumb node $\langle$`0x2dbfc`$, 1\rangle$, i.e., the node at the `0x2dbfc` row and N_T column, represents the Thumb `str` instruction at the same address. The inlined data starting at address `0x2dc20` are represented as two data nodes $\langle$`0x2dc20`$, -1\rangle$ and $\langle$`0x2dc22`$, -1\rangle$.

B. Edges

We introduce two kinds of edges: *implication edges* E_I and *conflict edges* E_C. The former is directed and the latter undirected. An implication edge is from a node to another if the former implies the latter. For example, the ARM node $\langle$`0x2dbfc`$, 0\rangle$ in Figure 6 implies the ARM node $\langle$`0x2dc00`$, 0\rangle$, denoted by the solid directed edge between the two, because if address `0x2dbfc` is decoded to an ARM instruction, the next address `0x2dc00` must be an ARM instruction too as the former does not change the mode.

A conflict edge is introduced from a node to another node if both cannot be true at the same time. For example, the ARM node $\langle$`0x2dbfc`$, 0\rangle$ in Figure 6 conflicts with the Thumb node $\langle$`0x2dbfc`$, 1\rangle$ and with the data node $\langle$`0x2dbfc`$, -1\rangle$, denoted by the dashed undirected edges among them. Implication edges are derived from the following three kinds of semantic information acquired from the previous stage.

1. Explicit Control Flow Transfer. For most non-branching and direct branching instructions, the next instruction and its mode are explicit. As such, implication edges are introduced from an instruction to its explicit control flow successor.

2. Implicit Control Flow Transfer. The static analysis may disclose indirect control transfer targets. For example, the target of `bx r0` at `0x2dc1c` in Figure 1(a) is known to be `0x2dc25` as discussed in Section IV, thus an implication edge is introduced in Figure 6 from node $\langle$`0x2dc1c`$, 0\rangle$ to $\langle$`0x2dc24`$, 1\rangle$.

3. Memory Access. An implication edge is added from an instruction node to a data node if the former accesses the latter.

TABLE III: Ten representative program properties used for counting the vertex weight

Type	Property	Explanation
Semantics	Memory dependency	A load instruction accesses the memory stored by a preceding store instruction.
	Array-like access	An instruction accesses memory in the form of "*base* + *index* × *scale*".
	Consecutive accesses	A set of instructions access consecutive addresses.
	Multi-level pointer dereference	A multi-level pointer dereference.
	Temporary variable loads	An instruction accesses a memory that is frequently accessed by others.
	Indirect jump target	An instruction is the target of some indirect jump.
Syntax	Register define-use	An instruction defines the value of a register and another instruction uses the register.
	String-like data	Consecutive data bytes constitute a null-terminated and human-readable string.
	Common jump target	Multiple direct jump instructions share the same target.

The edge set E of the graph is hence the following.

Definition V.2 (E)**.** $E = E_I \cup E_C$.

C. Node Weights

We discuss how the weight of each node is computed in the following. Observe that the more semantic behaviors an instruction exhibits, the more likely it is a true instruction. Ideally, the weight value of a node should reflect the number of semantic behaviors that it is involved in. To achieve this, we count the number of semantic relations derived by the static analysis in which the node is involved. Table III presents the semantic relations that we consider. The three columns present the type, either syntactic or semantic, the relation, and the explanation, respectively. In particular, we consider six semantic relations as follows. ***Memory dependency*** is a basic relation indicating a pair of load and store instructions access the same memory location. Note that such relations can be derived by comparing abstract values in address operands. ***Array-like access*** models the behavior of accessing an array element by a specific addressing expression (e.g., `p[i]` in Table I). The intuition is that false instructions are unlikely to have such expressions. Note that this relation can be easily determined by checking if an address operand has an affine abstract value. ***Consecutive accesses*** are a relation across instructions if they access consecutive addresses. It is common in memory traversal functions (e.g., `memcpy` and `strcpy`), but highly uncommon in false instructions. D-ARM also considers ***multi-level pointer dereference***, describing that a pointer is dereferenced multiple times (e.g., `int ***p`). Moreover, recall that ARM binaries tend to use temporary variables. This is reflected by frequent loads from the same memory location. We hence consider the ***temporary variable loads*** relation denoting such accesses. Finally, the ***indirect jump target*** relation describes instructions whose addresses are indirect jump targets, as disclosed by the static analysis. Note that different from P-Disasm, D-ARM's static analysis provides rich semantic constraints.

We also consider a few syntactic relations that can be derived without interpretation. The ***register define-use*** relation describes that an instruction defines a register and another

instruction uses the register. The ***string-like data*** relation denotes a sequence of consecutive null-terminated and human-readable date bytes, a common machine-level representation of constant strings. ***Common jump target*** models an instruction being the target of multiple direct control transfers.

Example. In Figure 1(a) the true ARM instruction at address `0x2dc18` is in a semantic memory dependency relation with the instruction at `0x2dc0c` (the green shadowed ones), and in two syntactic relations, i.e., the *register define-use* relation with the instructions at addresses `0x2dbfc` and `0x2dc1c`. Hence, its weight is 3. □

D. Graph Problem and Solution

With the definitions, we reduce ARM disassembly to a constrained node selection problem in graph theory. Specifically, each time a node is selected, two requirements should be satisfied. First, all the reachable nodes by implication edges should also be selected. Second, all the adjacent nodes by conflict edges must not be selected together. There may be more than one satisfying solutions. In this case, the selected nodes with the maximum total weight denote a solution with the richest semantics. We hence use that as the disassembly results. The formal definition of the problem is given as follows.

Definition V.3 ($\overline{\text{AD}}$)**.** *The ARM Disassembly task ($\overline{AD}$) is to find a subset $N' \subseteq N$ with the maximum total weight $p = \sum_{n_i \in N'} w(n_i)$ such that:*

- *For each node $n_i \in N'$, all its reachable nodes by E_I are also in N'. That is, $\forall n_i \rightarrow n_j \in E_I, n_j \in N'$ if $n_i \in N'$*
- *For each node $n_i \in N'$, all its adjacent nodes by E_C are not in N'. That is, $\forall n_i, n_j \in N', (n_i, n_j) \notin E_C$*

We reduce $\overline{\text{AD}}$ to a well-studied graph problem, *maximum weight independent set* ($\overline{\text{MWIS}}$) [17], and then leverage an existing algorithm to find the (approximately) optimal solution. $\overline{\text{MWIS}}$ tries to find a maximum-weighted subgraph G' whose nodes are not adjacent to each other in the original graph G.

Definition V.4 ($\overline{\text{MWIS}}$)**.** *Given an undirected graph $G = (N, E)$, an independent set of the graph is a set of nodes such that any pair of these nodes are not adjacent. That is, a subset $N' \subseteq N$ is an independent set if $\forall n_i, n_j \in N', (n_i, n_j) \notin E$. The Maximum Weight Independent Set problem ($\overline{MWIS}$) is to find the independent set with the maximum total weight in a node-weighted undirected graph.*

To reduce $\overline{\text{AD}}$ to $\overline{\text{MWIS}}$, we replace directed implication edges with undirected conflict edges. Intuitively, given an implication edge that node n_i implies node n_j, node n_i inherits all the conflicts of node n_j. Therefore, we replace the edge $n_i \rightarrow n_j$ with edges from n_i to all the conflicts of n_j (and of the other reachable nodes from n_i). This is illustrated by an example in Figure 7. The implication edge $a_0 \rightarrow a_4$ in red on the left is replaced with the two red dashed edges on the right.

Theorem V.1. *The optimal solution of $\overline{MWIS}$ is equivalent to the optimal solution of $\overline{AD}$.*

The proof of Theorem V.1 is given in Appendix B.

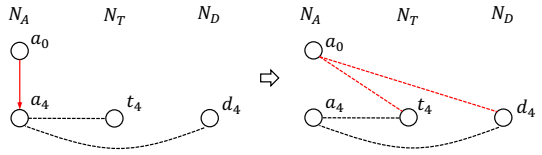


Fig. 7: Example of the graph transformation

$\overline{\text{MWIS}}$ is NP-hard. However, there are many approximate solutions [17], [24], [25]. Considering the efficiency requirement, we adopt a greedy algorithm [24], which iteratively selects a node of minimum weighted degree and removes it and its neighbors. It was proved that the algorithm can achieve a lower bound of $max(W/(\overline{d}_w+1), W/(\delta_w+1))$, where W is the total weight, $\overline{d}_w$ is the weighted average degree, and δ_w is the weighted inductiveness of the graph. Since the algorithm is not our contribution, details are elided.

VI. Evaluation

The evaluation of D-ARM addresses the following research questions:

- **RQ1:** How does D-ARM compare to existing disassemblers with respect to the precision, recall, and efficiency?
- **RQ2:** How does the static analysis improve the effectiveness of our disassembler?
- **RQ3:** Is D-ARM still effective when disassembling obfuscated binaries?
- **RQ4:** Can D-ARM benefit the downstream security application of binary rewriting?

A. Experiment Setup

The SPEC Dataset. As shown in Table IV, our experiments are conducted over a set of binaries built from SPEC CINT2000 [27] and SPEC CINT2006 [28], two standard benchmark suites widely used by prior works [29], [13]. Each of the two suites contains twelve programs. For the first two research questions, we build three groups of binaries, SPEC-Basic, SPEC-Data, and SPEC-Inter, using two compilers, i.e., GCC-5.5 and Clang-11, with five optimization levels, i.e., from `O0` to `O3`, and `Os`. The benchmark programs are compiled to binaries in two instruction sets, i.e., ARM and Thumb, with the compiling options `-marm` and `-mthumb`. Considering the various ARM architecture versions, we also choose two CPU architectures, i.e., ARMv5t and ARMv7a, where the Thumb instructions are 16-bit only in the former and 16/32-bit mixed in the latter. Thus, we have 40 combinations of compiling options and obtain totally 480 binaries for SPEC CINT2000 and SPEC CINT2006, respectively.

As discussed in Section III, inlined data and ARM/Thumb interleaving are the two main challenges for ARM disassembly and the default compilation configurations of SPEC (SPEC-Basic) are too simplistic to disclose the challenges. Real-world ARM binaries are usually used in commercial devices and applications, which are not public. It is difficult to acquire the ground truth information needed in controlled experiments. We hence generate two special SPEC datasets by different compiling strategies to showcase the challenges. First, due to

TABLE IV: SPEC dataset information

Dataset	Benchmark	Architecture	ISA	Compiler	# Binaries	# Bytes and Percentage of Inline Data
Basic	SPEC 2000 SPEC 2006	ARMv5t, ARMv7a	A32, T32	GCC-5.5, Clang-11	480 480	6,179,256 (4.6%) 14,246,186 (4.8%)
Data	SPEC 2000 SPEC 2006	ARMv5t, ARMv7a	A32, T32	GCC-5.5, Clang-11	480 480	21,187,922 (14.3%) 94,212,922 (25.0%)
Inter	SPEC 2000 SPEC 2006	ARMv5t, ARMv7a	A32, T32	GCC-5.5, Clang-11	480 480	6,188,036 (4.6%) 14,290,306 (4.8%)
AArch64	SPEC 2000 SPEC 2006	ARMv8a	A64	GCC-5.5, Clang-11	110 120	141,760 (0.5%) 439,920 (0.5%)
Obfuscation	SPEC 2000 SPEC 2006	ARMv5t, ARMv7a	A32, T32	Clang-11	720 717	135,713,528 (31.9%) 327,412,564 (33.2%)
AOSP[26]	Android libraries	aosp_arm-eng	A32, T32	Clang-6	669	7,173,708 (6.9%)

the limitation of small binary size, the binaries in SPEC-Basic may not contain substantial inlined data. Thus, we build SPEC-Data with some special processing. That is, after compiling every single source code file to an object file, we revise the object file by renaming the data section (i.e., the `.rodata` section) to `.text.xxx`, where `xxx` denotes a random character string. At link time, all data in the `.text.xxx` sections will be merged into the code section Thus, the produced binaries contain substantial code and data interleavings. The last column of Table IV shows that the binaries in SPEC-Data contain $3.5\times$ – $6.6\times$ inlined data compared to SPEC-Basic.

Second, when building a program, SPEC compiles all files with the same compilation options specified in its config file, which makes each binary in SPEC-Basic and SPEC-Data composed of instructions from a single instruction set, i.e., ARM by `-marm` or Thumb by `-mthumb`. However, a real application may compile different parts separately with different instruction sets, considering the demand for either high performance or better code density. Figure 13 (in Appendix) depicts the ARM and Thumb breakdown for binaries from real-world Android libraries, delineating the sheer quantity of instruction mode interleavings in real-world applications. For example, libneuralnetworks.so in AOSP is composed of 48% ARM and 46% Thumb. There are 6250 bx/blx instructions that could cause interleaving (3%). Thus, we also do some special processing to build SPEC-Inter. Specifically, given a set of source files, we compile them alternately using the options `-marm` and `-mthumb`. The object files with ARM and Thumb instructions are then linked together, producing binaries with more ARM/Thumb interleavings. Note that we still follow the normal compilation process without introducing additional data or code. Compared to SPEC-Basic and SPEC-Data, the binaries in SPEC-Inter have a better balance between ARM and Thumb instructions. Both SPEC-Data and SPEC-Inter use the same combination of compiling options as SPEC-Basic and thus the same number of binaries as shown in Table IV. More discussion about SPEC-Data and SPEC-Inter can be found in our supplementary material [18].

In addition to the 32-bit architecture ARMv5t and ARMv7a, we also build another group of binaries compiled for ARMv8a (SPEC-AArch64), which is a 64-bit architecture and uses the A64 instruction set. Except one program in SPEC CINT2000 that fails to be built for ARMv8a, we have 230 binaries in this dataset. With these binaries, we compare our approach to a recent disassembler, D-Disam [15], which only supports 64-bit ARM binaries.

For the third research question, we follow a seminal work on obfuscation against disassembly [30] to obfuscate the binaries in SPEC-Basic. The obfuscation works by injecting random junk bytes after each direct branch and after probabilistically selected non-branch instructions. The bytes are injected in a way that they never get executed. We use three obfuscation levels, i.e., $r = 0\%$, 50%, and 100%, denoting the probabilities when selecting non-branch instructions, where the higher level indicates more inserted junk bytes. Note that the obfuscation technique used here is different from typical obfuscators, e.g., O-LLVM [31], which aim to increase the difficulty of decompilation instead of thwarting disassembly. Also, it was shown that O-LLVM does not incur much trouble for disassemblers [13], [30]. We implement the obfuscator on LLVM. Thus, this experiment is only carried out with Clang, not GCC.

Real-world Android Binaries. Besides the SPEC datasets, we also evaluate our tool on a public dataset of real-world ARM binaries [26]. It contains 667 Android libraries built from the Android Open Source Project version 9.0.8 with the default target device option aosp_arm-eng [32].

Ground truth is collected based on mapping symbols[26].

Implementations and Baselines. In D-ARM, we use Capstone [33] to generate the superset instructions. Our superset instruction interpretation is implemented based on radare2 [34]. We select two state-of-the-art disassemblers, P-Disasm and XDA, as the representatives of rule-based methods and ML-based methods, respectively. P-Disasm only supports x86 and cannot be used for ARM directly [29]. We hence implement an ARM version of P-Disasm. As the models provided by XDA are trained on x86/x64 binaries, we strictly follow its settings and train two models, one on SPEC-Basic

TABLE V: Precision (P, %), recall (R, %), and the F1 score (%) of disassembling AArch32 binaries

Instructions

Dataset	Benchmark	Ghidra			IDA			P-Disasm			XDA			D-ARM		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Basic	SPEC 2000	99.97	93.39	96.44	99.98	97.27	98.58	93.13	93.41	92.81	99.98	99.99	**99.99**	99.99	99.97	99.98
	SPEC 2006	99.72	89.73	94.13	99.93	97.30	98.53	94.45	94.68	94.25	99.80	99.93	99.86	99.92	99.82	**99.87**
Data	SPEC 2000	99.65	92.96	96.07	99.72	97.05	98.34	80.05	91.54	85.06	90.69	99.99	94.98	98.61	98.48	**98.53**
	SPEC 2006	99.18	89.37	93.74	99.54	96.32	97.83	86.92	94.28	90.07	89.25	99.92	94.17	98.06	98.26	**98.14**
Inter	SPEC 2000	99.97	92.61	95.98	99.95	96.36	98.08	89.77	88.43	88.54	99.93	99.93	99.93	99.95	99.90	**99.93**
	SPEC 2006	99.96	89.54	94.17	99.92	96.50	98.13	91.57	90.70	90.66	99.70	99.73	99.72	99.84	99.67	**99.76**
AOSP	Libraries	99.75	92.34	95.90	99.96	98.90	99.43	85.91	86.45	86.18	99.57	99.75	99.66	99.79	99.86	**99.82**

Reachable Blocks

Dataset	Benchmark	Ghidra			IDA			P-Disasm			XDA			D-ARM		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Basic	SPEC 2000	98.45	80.25	88.41	91.74	90.45	91.09	82.33	88.16	85.14	99.32	99.84	**99.58**	99.27	99.77	99.52
	SPEC 2006	93.34	65.09	76.69	90.78	89.83	90.29	73.79	78.92	76.26	95.16	98.22	96.67	97.75	98.98	**98.36**
Data	SPEC 2000	90.12	80.17	84.78	91.21	89.28	90.24	80.05	87.97	83.81	69.20	99.75	81.69	91.10	97.04	**93.93**
	SPEC 2006	90.41	64.83	75.50	90.21	89.01	89.59	68.26	79.84	73.59	84.33	98.02	90.59	87.35	95.99	**91.33**
Inter	SPEC 2000	98.53	80.33	88.48	91.42	89.24	90.32	63.23	80.18	70.64	95.35	98.97	97.12	98.59	99.08	**98.83**
	SPEC 2006	96.74	64.84	77.60	90.57	88.80	89.66	49.97	74.75	59.82	88.13	96.10	91.94	96.48	97.74	**97.11**
AOSP	Libraries	72.48	74.05	73.26	70.59	79.60	74.83	68.45	73.47	70.87	87.56	94.94	91.10	95.10	97.42	**96.25**

2000 and the other on SPEC-Basic 2006. We use the former to evaluate the datasets built on SPEC 2006 and the latter for the datasets built on SPEC 2000. We also compare D-ARM with IDA Pro 7.5.2 and Ghidra 10.0.1.

Evaluation Metrics. We evaluate the efficacy of different disassemblers using three metrics, i.e., precision, recall, and the F1 score, at two different granularities, i.e., the granularity of instructions and the granularity of reachable blocks. Here, a reachable block means a reachable control flow sub-graph following explicit edges. Note that if a disassembly technique cannot identify the entry point of a sub-graph, it misses the entire reachable sub-graph. This metric hence aims to count the missing sub-graphs. Precision is the ratio of the number of correctly disassembled instructions/blocks to the total number of disassembled instructions/blocks. Recall is the ratio of the number of correctly disassembled instructions/blocks to the total number of true instructions/blocks. F1 is computed from the two.

Environment. All the experiments are run on a server equipped with a 48-cores CPU (Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz) and 188G memory. we set a fixed timeout of 2 hours for the disassembly of each program.

B. RQ1: Accuracy and Efficiency

1) Comparing D-ARM to Ghidra, IDA, P-Disasm, and XDA: Table V shows the precision, recall and the F1 scores of the state-of-the-arts as well as our approach on the SPEC datasets and AOSP. The highest F1 scores in each row are highlighted. We can observe that D-ARM outperforms other tools in almost all settings except for XDA on the simplest dataset, SPEC2000-Basic, on which both XDA and D-ARM achieve an F1 score close to 100%. Note that, although XDA achieves high recall on SPEC-Basic and SPEC-Inter at the instruction level, its precision degrades a lot on SPEC-Data, which shows that XDA disassembles data bytes as instructions. Also, its performance on reachable blocks proves that XDA may not be able to consider sufficient semantics when making predictions, as we discussed in Section III-B.

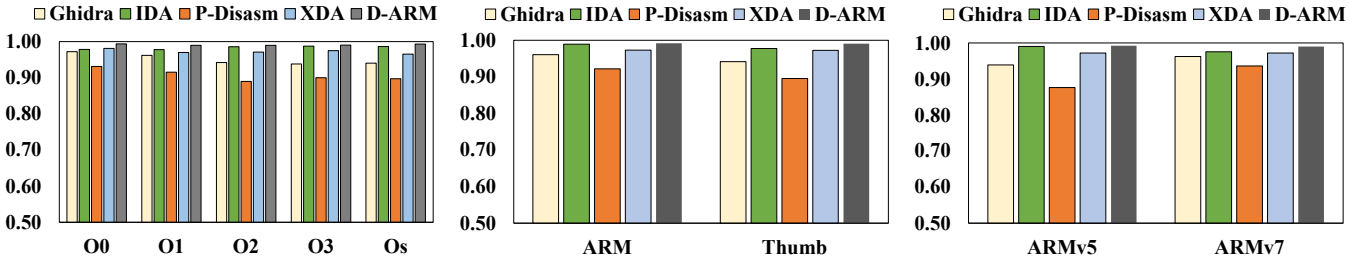
IDA and Ghidra have higher precision than recall, which implies that they are better because they do not interpret data as instructions. Actually, IDA utilizes a large set of manually crafted rules in disassembly and is closed source. Based on our experience and observations, IDA combines recursive and linear disassembly. It finds function entries and then follows control flow to disassemble, which leads to high precision. However, if a function entry is not successfully identified, e.g., one can only be reached by indirect branches, it will generate false negatives and degrade the recall. The strategy of starting from function entries performs well on SPEC-Data, where the inlined data lie between functions. However, in complex binaries, e.g., the obfuscation discussed in Section VI-D, when there are more inlined data within functions, IDA decodes data as instructions and its precision degrades significantly. D-ARM can deal with these challenging cases by solving the implication constraints and conflict constraints in the graph model. We have similar observations on Ghidra, while its recall and F1 are lower than IDA and our approach.

In addition, we perform a sensitivity study regarding compilation options, architecture versions, and instruction modes, using SPEC-Basic and SPEC-Data. The results are shown in Figure 8. D-ARM is insensitive to these settings and consistently the best performing tool.

2) Comparing D-ARM to D-Disasm and Spedi: We also compare D-ARM with two other disassemblers, D-

TABLE VI: F1 scores of recovering instruction boundaries and reachable blocks on obfuscated binaries

Dataset	Obf. Level	Instruction Boundary F1 (%)					Reachable Blocks F1 (%)				
		Ghidra	IDA	P-Disasm	XDA	D-ARM	Ghidra	IDA	P-Disasm	XDA	D-ARM
SPEC 2000	r=0	96.95	90.69	88.37	89.03	**99.39**	88.94	33.66	51.27	26.16	**96.89**
	r=50	94.80	77.67	82.62	68.71	**98.82**	57.01	16.02	17.57	7.28	**94.25**
	r=100	94.84	70.64	74.40	56.11	**97.74**	46.57	11.71	10.10	4.00	**88.72**
SPEC 2006	r=0	93.30	89.75	91.95	86.41	**99.22**	78.88	32.02	32.59	19.03	**96.45**
	r=50	93.63	76.63	83.66	63.73	**98.17**	76.88	16.04	11.28	5.86	**91.10**
	r=100	92.73	70.94	77.34	48.65	**95.94**	56.45	12.75	8.35	2.83	**78.16**



(a) (b) (c)

Fig. 8: The F1 scores (Y-axis) at different optimization levels (X-axis in (a)), for different instruction sets (X-axis in (b)), and under different architectures (X-axis in (c)).

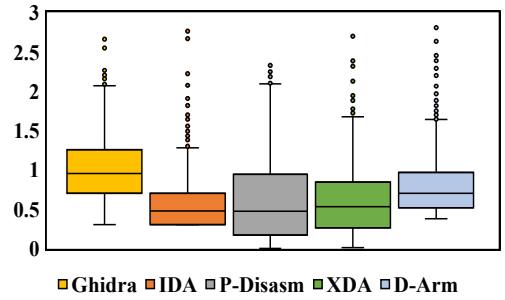


Fig. 9: Time cost taken by different tools. Each value on Y-axis is in log scale: $\log_{10}(\text{time in seconds} + 1)$.

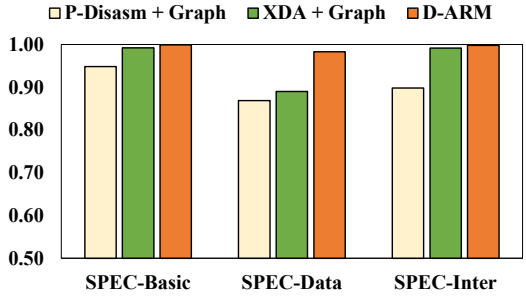


Fig. 10: Results of the ablation study: the F1-scores (Y-axis) obtained over different datasets (X-axis).

Disasm [15] and Spedi [35], which could only be able to disassemble partial of our datasets. The evaluation results are shown in Appendix C.

3) Efficiency: Figure 9 shows the time cost taken by D-ARM and the baseline approaches on the AOSP dataset, which is composed of real-world binaries. All the binaries (with the largest 8.9MB) take D-ARM less than 650 seconds to disassemble. Note that P-Disasm fails in disassembling more than half of the binaries due to its high memory cost. The failure rates are shown in the supplementary material [18].

C. RQ2: Effectiveness of the Static Analysis

To measure the effectiveness of our static analysis, we conduct an ablation study where we disable the static analysis from D-ARM and replace the instruction weights with those produced by P-Disasm and XDA. We use P-Disasm+Graph and XDA+Graph to denote the approaches where the interpretation is replaced. Figure 10 shows the F1 scores of running P-Disasm+Graph, XDA+Graph, and D-ARM over the three datasets, SPEC-Basic, SPEC-Data, and SPEC-Inter. Observe that D-ARM performs the best — the F1 score is always close to 100%. In contrast, the F1 scores of both P-Disasm+Graph and XDA+Graph degrade to below 90% on SPEC-Data and P-Disasm+Graph degrades to below 90% on SPEC-Inter. This indicates the importance of our static analysis.

D. RQ3: Effectiveness on Obfuscated Code

Obfuscating binary code is one of the main methods to evade disassemblers. In this experiment, we evaluate how well the tools can penetrate obfuscation. Table VI shows the evaluation results (F1 scores) on the obfuscated code. At the instruction level, the F1 score of D-ARM is consistently higher than 95% and up to 99.39%, which is the best among all disassemblers. In contrast, the F1 scores of most other tools drop below 90%, even less than 50%, exhibiting weak resistance to obfuscation. At the block level, the advantage of D-ARM is more apparent as our F1 score is 9% to 2118% higher than other tools. As an example, the F1 score of Ghidra, the best performing tool among recent works, drops below 50% on SPEC 2000 ($r = 100$) while our F1 score is 88.72%. This is because D-ARM is semantics oriented, due to its instruction interpretation component.

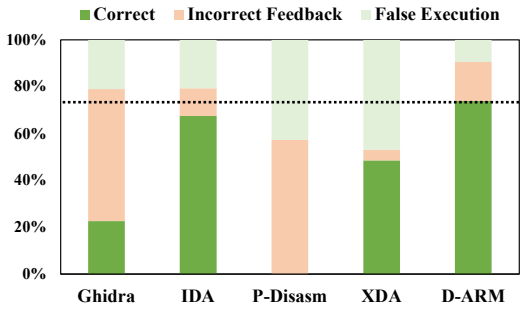


Fig. 11: Percentages (Y-axis) of execution failure, incorrect feedback, and correct binaries after rewriting with different tools (X-axis).

E. RQ4: A Case Study on Binary Rewriting

We also conducted a case study to demonstrate the potential improvement that our tool can bring to downstream security applications. Specifically, static binary instrumentation plays a crucial role in many security scenarios including binary-only fuzzing [36], dynamic taint analysis [37], and COTS program hardening [38]. *Pathcerex* [39] is a state-of-the-art ARM-compatible instrumentation tool, originally developed for DARPA's Cyber Grand Challenge [40]. In the study, we replace the underpinning disassembler in *Patcherex* with D-ARM to perform static binary instrumentation. We evaluate the new *Patcherex* on the SPEC2000 binaries and the real-world Android daemons. The results show that the D-ARM-based *Patcherex* has greater potential to safely rewrite binaries and collect precise runtime information for fuzzing, compared with the counterparts driven by other disassembly tools.

Patcherex adapts a trampoline-based instrumentation method. That is, at each patch point, *Patcherex* patches a set of instructions to detour the control flow to another code region. The code region can be arbitrarily manipulated, and hence the instrumentation code is executed there. The control flow is detoured back after that. In this study, we leverage *Patcherex* to realize the instrumentation of AFL. AFL instruments a piece of code before every basic block to monitor the path coverage, which further guides seed mutation and, hence, is essential to gray-box fuzzing. The instrumentation is built upon the assumption that the upstream disassembler can correctly identify all the instructions. However, such an assumption may not hold in practice due to the limitation of existing disassemblers described in Section III-B. Intuitively, incorrect disassembly results can lead to both incorrect path coverage (e.g., some basic blocks are not monitored due to false negatives of disassembly) and execution failures (e.g., trampoline code in ARM is patched inside Thumb or data regions).

We conduct experiments on two datasets, the SPEC2000 programs and the Android daemons. Note that all binaries in the two datasets are produced with their default build system without any change of the compilation tool-chains.

SPEC. First, we instrument the SPEC2000 programs, i.e., the SPEC-Basic dataset used in Section VI-A, with the disassembly results provided by different disassemblers. Then, we test the rewritten binaries with all the test cases provided by SPEC. According to the execution status and outputs, each binary is labeled as one of the following.

1. Execution Failure. The binary is considered to fail in execution if it crashes or generates inconsistent outputs with the standard outputs provided by SPEC.

2. Incorrect Feedback. If the binary executes correctly, we then compare the collected path coverage with the coverage by instrumenting the ground-truth binaries (i.e., those correctly disassembled). If they are different, the instrumented binary is considered to provide incorrect feedback.

3. Correct. Otherwise, the binary is considered correct.

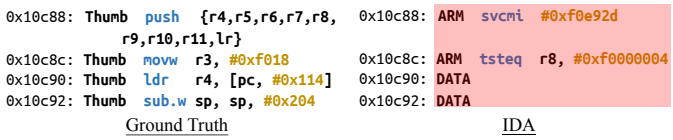
In Figure 11, we show the percentages of the three types of instrumented binaries when using the disassembly results provided by different tools. We can see that, rewriting with D-ARM can generate the most correct binaries (the dark green bar) and also the fewest execution failures (the light green bar). Note that IDA and XDA have fewer binaries of incorrect feedback (the pink bar) because many binaries have already failed in execution and will not be tested/counted for incorrect feedback (no feedback at all). In an extreme case, the superset disassembly will have a 100% false execution and zero incorrect feedback.

TABLE VII: Rewriting results for binaries compiled with O3+mthumb+v7a

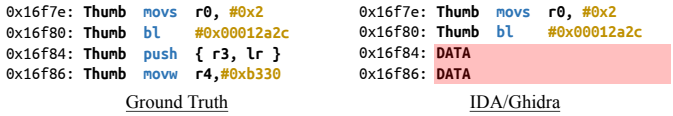
Binary	**Ghidra**		**IDA**		**P-Disasm**		**XDA**		**D-ARM**	
	Exe	FB	Exe	FB	Exe	FB	Exe	FB	Exe	FB
164.gzip	✓	✗	✓	✗	✗	-	✗	-	✓	✓
175.vpr	✓	✓	✗	-	✓	✗	✗	-	✓	✓
176.gcc	✓	✗	✗	-	✓	✗	✗	-	✗	-
181.mcf	✓	✗	✓	✗	✗	-	✓	✓	✓	✓
186.crafty	✓	✗	✗	-	✓	✗	✗	-	✓	✓
197.parser	✓	✗	✓	✓	✓	✗	✗	-	✓	✗
252.eon	✗	-	✗	-	✓	✗	✗	-	✗	-
253.perlbmk	✗	-	✗	-	✓	✗	✗	-	✓	✗
254.gap	✗	-	✓	✓	✓	✗	✗	-	✓	✓
255.vortex	✓	✗	✗	-	✓	✗	✗	-	✓	✓
256.bzip2	✓	✗	✓	✗	✗	-	✗	-	✓	✓
300.twolf	✓	✗	✗	-	✗	-	✗	-	✓	✓
Success Rate	0.75	0.08	0.42	0.17	0.67	0.00	0.08	0.08	**0.83**	**0.67**

In Table VII, we show the detailed results of a group of binaries with the compiling options of O3 and mthumb, which are usually more challenging than other options. We can see that D-ARM has the highest success rate (i.e., passing both checks). D-ARM fails in the executions of gcc and eon. Ghidra and P-Disasm pass the execution checking for the two binaries as they miss a lot of instructions and only provide a few block entries for rewriting. With less rewriting, the binary is highly likely to execute normally, while incorrect path coverage will be collected. Also, unlike the (good) overall success rate shown in Fig 11, IDA has a low success rate in this group of binaries. This is because rewriting is very sensitive to disassembly errors, and a small number of errors may cause failures. IDA has good results on ARM instructions but generates more errors on Thumb, as shown in Fig 8(b).

To better understand the influences of disassembly on



(a) A snippet from 186.crafty leads to execution failures



(b) A snippet from 256.bzip2 leads to incorrect feedback

Fig. 12: Examples of errors in rewriting cased by incorrect disassembly results

rewriting, we further investigate and show some failure cases from the binaries in Table VII. Fig 12a shows the start of the main function which should be four Thumb instructions. However, IDA mistakes them as two ARM instructions followed by data. Given the disassembly results, the trampoline code patched at address `0x10c88` contains ARM instructions. Then the binary fails in execution due to the wrong execution mode. In Fig 12b, both IDA and Ghidra miss the function entry at the address `0x16f84`, which is only reached by indirect branches. Then during rewriting, `0x16f84` is not considered as the block entry and no instrumentation is conducted. Although the execution is successful, the path coverage is incorrect. Note that for both examples, crafty and bzip2, the instrumented binaries based on D-ARM are correct.

Android Daemons. Besides SPEC, we also study a set of Android daemons, as these real-world ARM binaries may contain more instruction set interleavings and inlined data.

Unlike SPEC programs, Android daemons usually do not have standard input or output, and many of them usually work as background processes, which makes it hard to automatically test them. Thus, we randomly select 10 well-known daemons whose execution status could be observed directly or probed by executing some applications, which require the service provided by the daemons. The daemons we test are listed in the first column in Table VIII.

In this evaluation, we use a rooted Nexus-6p with Android 9. After rewriting a daemon, we replace the original one in the phone with the generated binary, reboot, and then run the daemon or its corresponding applications. Specifically, audioserver, cameraserver, and mediaserver are started automatically during the boot process. As such, an invalid audioserver makes the boot process failed. Systems can boot correctly with an invalid cameraserver or mediaserver, while opening the corresponding applications, e.g., the camera app, can trigger a crash. Other daemons could be executed directly and their execution status can be observed.

As shown in Table VIII, except screenrecord and sqlite3, which are challenging for all disassemblers, the instrumented daemons based on D-ARM do not report any error, while Ghidra, P-Disasm, and XDA fail for all daemons. This proves that, compared with other disassemblers, D-ARM provides much more accurate disassembly results for downstream applications such as rewriting even on real-world binaries.

TABLE VIII: Results of rewriting Android daemons

Binary	Ghidra	IDA	P-Disasm	XDA	D-ARM
atrace	✗	✗	✗	✗	✓
dumsys	✗	✓	✗	✗	✓
bootanimation	✗	✗	✗	✗	✓
reboot	✗	✓	✗	✗	✓
ping	✗	✓	✗	✗	✓
screenrecord	✗	✗	✗	✗	✗
sqlite3	✗	✗	✗	✗	✗
audioserver	✗	✗	✗	✗	✓
cameraserver	✗	✓	✗	✗	✓
mediaserver	✗	✓	✗	✗	✓

VII. Related Work

Disassembly is a critical step for understanding closed-source code and can be done via static [15], [13], [14] and dynamic [41] techniques. Besides the state-of-the-art disassemblers [13], [14], [11], [12], [35] we have extensively discussed in Section III-B, there are also some other tools studied in previous works. Objdump [16], PSI [42], and Uroboros [43] use linear sweep for disassembly, while Dyninst [44], Angr [45], BAP [46], and Radare2 [34] are based on recursive traversal disassembly. Besides the two basic disassembly strategies, different tools usually adopt different sets of algorithms or heuristics to improve the results. There have been a lot of systematization studies qualitatively or/and quantitatively evaluating these tools [20], [47], [48], [49], [50]. However, most previous works on disassembly focus on x86 binaries [15], [13], [14]. M. Jiang et al [26] empirically investigated existing disassemblers for ARM binaries. The study demonstrates that recent works often cannot address ARM-specific challenges induced by substantial inlined data and ARM/Thumb interleavings. Our work takes the first step to address both challenges and the evaluation results are promising.

VIII. Conclusion

We propose a novel method for ARM disassembly. It leverages a lightweight static analysis that interprets all the superset instructions, i.e., all the possible instructions (in different modes) for each address and generates initial information for instruction modes. It then models the program with a graph and reduces the disassembly problem to a maximum weight independent set problem, which can be solved using an existing approximate algorithm. Our system D-ARM substantially outperforms five state-of-the-art disassemblers.

Acknowledgment

We thank the anonymous reviewers for their valuable comments and suggestions. This research was supported, in part by DARPA VSPELLS - HR001120S0058, NSF1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] "Smartphone processors," https://www.arm.com/solutions/mobile-computing/smartphones.

[2] "Fugaku," https://tinyurl.com/3sy8hus3.

[3] A. Darki, M. Faloutsos, N. Abu-Ghazaleh, M. Sridharan *et al.*, "Idapro for iot malware analysis?" in *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, 2019.

[4] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on arm," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[5] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," *arXiv preprint arXiv:2105.05445*, 2021.

[6] C. Zhang, Y. Wang, and L. Wang, "Firmware fuzzing: The state of the art," in *12th Asia-Pacific Symposium on Internetware*, 2020.

[7] L. Di Bartolomeo, "Armwrestling: efficient binary rewriting for arm," Master's thesis, ETH Zurich, 2021.

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

[9] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "Razor: A framework for post-deployment software debloating," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[10] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019.

[11] "Ida pro," https://hex-rays.com/ida-pro/.

[12] "Ghidra," https://ghidra-sre.org/.

[13] K. Pei, J. Guan, D. Williams-King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[14] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019.

[15] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[16] "The gnu binary utilities - objdump," https://web.mit.edu/gnu/doc/html/binutils_5.html.

[17] S. Sakai, M. Togasaki, and K. Yamazaki, "A note on greedy algorithms for the maximum weighted independent set problem," *Discrete applied mathematics*, vol. 126, no. 2-3, 2003.

[18] "D-arm," https://github.com/yapengye/D-ARM.

[19] "Armv4t," http://ww1.microchip.com/downloads/en/DeviceDoc/DDI0029G_7TDMI_R3_trm.pdf.

[20] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[21] E. Bauman, Z. Lin, K. W. Hamlen *et al.*, "Superset disassembly: Statically rewriting x86 binaries without heuristics." in *NDSS*, 2018.

[22] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977.

[23] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *International conference on compiler construction*. Springer, 2004.

[24] A. Kako, T. Ono, T. Hirata, and M. M. Halldórsson, "Approximation algorithms for the weighted independent set problem," in *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 2005.

[25] S. Lamm, C. Schulz, D. Strash, R. Williger, and H. Zhang, "Exactly solving the maximum weight independent set problem on large real-world graphs," in *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2019.

[26] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren, "An empirical study on arm disassembly tools," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[27] J. L. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, no. 7, 2000.

[28] ——, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, 2006.

[29] "Probabilistic disassembly," https://github.com/KennethAdamMiller/superset_disassembler.

[30] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*, 2003.

[31] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm–software protection for the masses," in *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 2015.

[32] "Android open source project," https://source.android.com/.

[33] "The ultimate disassembler," https://www.capstone-engine.org/.

[34] "Radare2," https://github.com/radareorg/radare2.

[35] M. A. B. Khadra, D. Stoffel, and W. Kunz, "Speculative disassembly of binary code," in *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*. IEEE, 2016.

[36] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[37] S. Chen, Z. Lin, and Y. Zhang, "Selectivetaint: Efficient data flow tracking with static binary rewriting," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[38] P. Kiaei, C.-B. Breunesse, M. Ahmadi, P. Schaumont, and J. Van Woudenberg, "Rewrite to reinforce: Rewriting the binary to apply countermeasures against fault injection," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021.

[39] "patcherex," https://github.com/angr/patcherex.

[40] "2016 cyber grand challenge," https://en.wikipedia.org/wiki/2016_Cyber_Grand_Challenge.

[41] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[42] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A platform for secure static binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2014.

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

[44] "Dyninst," https://www.dyninst.org/.

[45] 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 *IEEE Symposium on Security and Privacy*, 2016.

[46] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011.

[47] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

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

[49] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, "From hack to elaborate technique—a survey on binary rewriting," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, 2019.

[50] R. Paleari, L. Martignoni, G. Fresi Roglia, and D. Bruschi, "N-version disassembly: differential testing of x86 disassemblers," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010.

[51] Y. Smaragdakis and M. Bravenboer, "Using datalog for fast and easy program analysis," in *International Datalog 2.0 Workshop*. Springer, 2010.

TABLE IX: Instruction Interpretation Example

Step	pc	Assembly Code	Trace	Actions		
				RS	MS	Jump Target
1	$\langle$0x2dbfc,0$\rangle$	**add** r5, r7, r8	$r5 := r7 + r8$	$[r5] = AD^{\langle 0x2dbfc,0\rangle}$	-	-
2	$\langle$0x2db00,0$\rangle$ $\langle$0x2db04,0$\rangle$	**movw** r6, #0xdc25 **movt** r6, #0x2	$r6 := 0x2dc25$	$[r6] = 0x2dc25$	-	-
3	$\langle$0x2db08,0$\rangle$	**mov** r1, r5	$r1 := r5$	$[r1] = AD^{\langle 0x2dbfc,0\rangle}$	-	-
4	$\langle$0x2db0c,0$\rangle$	**str** r6, [r1]	$\mathbf{W}(r1, r6)$	-	$[AD^{\langle 0x2dbfc,0\rangle}] := 0x2dc25$	-
5	$\langle$0x2db10,0$\rangle$	**sub** r6, #5	$r6 := r6 - 5$	$[r6] = 0x2dc20$	-	-
6	$\langle$0x2db14,0$\rangle$	**ldr** r6, [r6]	$r6 := \mathbf{R}(r6)$	$[r6] = 0xf898681b$	-	-
7	$\langle$0x2db18,0$\rangle$	**ldr** r0, [r5]	$r0 := \mathbf{R}(r5)$	$[r0] = 0x2dc25$	-	-
8	$\langle$0x2db1c,0$\rangle$	**bx** r0	**i-goto**$(r0)$	-	-	$\langle$0x2db24,1$\rangle$
9	$\langle$0x2db24,1$\rangle$	**push** {r3, lr}	-	-	-	-

APPENDIX

A. Running Example of the Superset Instruction Interpretation

Consider the example in Table IX which is derived from the code snippet in Figure 1(a) with all conditional suffixes removed. The interpretation step, *pc*, assembly code, trace in our language, and the interpretation actions are shown in the columns from left to right, respectively. Since registers $r7$ and $r8$ hold an unknown value, a symbolic value $AD^{\langle 0x2dbfc,0\rangle}$ is assigned to $RS[r5]$. In the second step, a constant 0x2dc25 is assigned to $RS[r6]$. Register $r1$ inherits the abstract value $AD^{\langle 0x2dbfc,0\rangle}$ from $r5$ in the next step. The fourth step stores the value in register $r6$ into the memory denoted by $[r1]$. Specifically, 0x2dc25 is written to the memory $MS[AD^{\langle 0x2dbfc,0\rangle}]$. At step 5, register $r6$ is updated to 0x2dc20 and a pointer deference of it then takes place at step 6. Note that the abstract value of register $r6$ is a constant so that D-ARM read the data bytes from the corresponding virtual address. As shown in Figure 1(a), $MS[0x2dc20] =$ 0xf898681b. Step 7 sets the value of $r0$ to the data in $[r5]$. Observe that the value of $r5$ is $AD^{\langle 0x2dbfc,0\rangle}$ which matches a preceding memory store at step 4, and $r0$ is hence set accordingly. At step 8, the interpreter determines that the indirect jump target $r0$ holds a constant 0x2db25 with a least significant bit 1, and hence switches the instruction mode to Thumb and jumps to 0x2db24.

B. Proof of Theorem V.1

First, we give a detailed definition of the transformation from $\overline{\text{AD}}$ to $\overline{\text{MWIS}}$ by replacing directed edges E_I with undirected edges $E_{I\to C}$.

Predicate A.1 (R)**.** *$R(n_i, n_j)$ denotes if n_j is reachable from n_i by E_I. An inductive definition of predicate R is as follows,*

1) $\forall n_i \in N, R(n_i, n_i)$.

2) $\forall n_i, n_j, n_k \in N,\ R(n_i, n_j) \wedge \langle n_j, n_k\rangle \in E_I \to R(n_i, n_k)$

Predicate R is reflexive and transitive by its definition. That is, $\forall n_i, n_j, n_k \in N$ it must satisfy:

- Reflexivity: $R(n_i, n_i)$, i.e., every node is reachable from itself.
- Transitivity: if $R(n_i, n_j) \wedge R(n_j, n_k)$, then $R(n_i, n_k)$.

Definition A.1 (T)**.** *$T : G_{\overline{AD}} \to G_{\overline{MWIS}}$, denoting the transformation from $G_{\overline{AD}} = (N, E_I, E_C)$ to $G_{\overline{MWIS}} = (N, \varnothing, E_C \cup E_{I\to C})$ by replacing directed edges E_I with undirected edges $E_{I\to C}$. Specifically, $\forall n_{a_s}, n_{a_t}, n_{b_s}, n_{b_t} \in N$, s.t. $(n_{a_t}, n_{b_t}) \in E_C \wedge R(n_{a_s}, n_{a_t}) \wedge R(n_{b_s}, n_{b_t}), (n_{a_s}, n_{b_s}) \in E_{I\to C}$.*

Then Theorem V.1 can also be defined as follows.

Theorem A.1. *Assuming the optimal solution of $\overline{AD}$ in $G_{\overline{AD}}$ is $N'_{\overline{AD}}$, and the one of $\overline{MWIS}$ in $G_{\overline{MWIS}} = T(G_{\overline{AD}})$ is $N'_{\overline{MWIS}}$, $N'_{\overline{AD}} = N'_{\overline{MWIS}}$.*

The proof of Theorem A.1 is given as follows.

Proof.

1) First, we prove $N'_{\overline{\text{AD}}}$ is also an independent set of $G_{\overline{\text{MWIS}}}$. $\forall n_i, n_j \in N'_{\overline{\text{AD}}}$,
   a) According to the definition of $\overline{\text{AD}}$, as $n_i, n_j \in N'_{\overline{\text{AD}}}$, we have $(n_i, n_j) \notin E_C$.
   b) Assume $(n_i, n_j) \in E_{I\to C}$. According to the definition of T, $\exists n_{a_t}, n_{b_t}$, s.t. $(n_{a_t}, n_{b_t}) \in E_C \wedge R(n_i, n_{a_t}) \wedge R(n_j, n_{b_t})$.
   As $n_i \in N'_{\overline{\text{AD}}}$ and $R(n_i, n_{a_t})$, n_{a_t} must be in $N'_{\overline{\text{AD}}}$. Similarly, n_{b_t} is also in $N'_{\overline{\text{AD}}}$.
   Thus, both n_{a_t} and n_{b_t} are in $N'_{\overline{\text{AD}}}$. We can have $(n_{a_t}, n_{b_t}) \notin E_C$. Contradiction.
   Therefore, $(n_i, n_j) \notin E_{I\to C}$
   c) According to a) and b), $\forall n_i, n_j \in N'_{\overline{\text{AD}}}$, $(n_i, n_j) \notin E_C$ and $(n_i, n_j) \notin E_{I\to C}$. Thus, $N'_{\overline{\text{AD}}}$ is also an independent set of $G_{\overline{\text{MWIS}}}$.
2) Second, we prove $N'_{\overline{\text{MWIS}}}$ is also a solution of $\overline{\text{AD}}$ in $G_{\overline{\text{AD}}}$.
   a) $\forall n_i, n_j \in N'_{\overline{\text{MWIS}}}$, according to the definition of $\overline{\text{MWIS}}$, $(n_i, n_j) \notin E_C$.
   b) $\forall n_i \in N'_{\overline{\text{MWIS}}}$, we are going to prove that $\forall n_j \in N$ s.t. $\langle n_i, n_j\rangle \in E_I$, $n_j \in N'_{\overline{\text{MWIS}}}$ by contradiction. We assume $\exists n_j$ s.t. $\langle n_i, n_j\rangle \in E_I$ and $n_j \notin N'_{\overline{\text{MWIS}}}$.
      i) $\forall n_k$ s.t. $(n_j, n_k) \in E_C$, as $R(n_i, n_j)$ and $R(n_k, n_k)$, according to the definition of T, we can have $(n_i, n_k) \in E_{I\to C}$. Thus, $n_k \notin N'_{\overline{\text{MWIS}}}$.
      ii) $\forall n_k$ s.t. $(n_j, n_k) \in E_{I\to C}$, according to the definition of T, $\exists n_{j_t}, n_{k_t} \in N$, s.t. $(n_{j_t}, n_{k_t}) \in E_C \wedge R(n_j, n_{j_t}) \wedge R(n_k, n_{k_t})$. As $R(n_i, n_j) \wedge R(n_j, n_{j_t})$, we can also have $R(n_i, n_{j_t})$ (i.e., transitivity of R), which implies $\exists n_{j_t}, n_{k_t} \in N$, s.t. $(n_{j_t}, n_{k_t}) \in E_C \wedge$

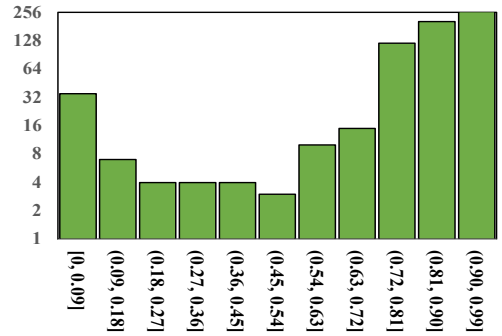


Fig. 13: Distribution of the percentage of Thumb instructions of Android libraries: the number of Android binaries (Y-axis in log scale) with certain percentage (X-axis) of the Thumb instructions.

TABLE X: Comparing D-ARM with D-Disasm in Precision (P), Recall (R), and F1 scores (%) over the 64-bit binaries

Instructions							
Dataset		**D-Disasm**			**D-ARM**		
		P	R	F1	P	R	F1
AArch64	SPEC2000	100.00	93.16	96.50	100.00	100.00	100.00
	SPEC2006	100.00	95.12	97.50	100.00	100.00	100.00
Reachable Blocks							
Dataset		**D-Disasm**			**D-ARM**		
		P	R	F1	P	R	F1
AArch64	SPEC2000	99.80	88.88	94.00	99.98	99.99	99.99
	SPEC2006	99.57	86.70	92.70	99.91	99.96	99.93

$R(n_i, n_{j_t}) \wedge R(n_k, n_{k_t})$. Thus, $(n_i, n_k) \in E_{I\to C}$ by the definition of T, and $n_k \notin N'_{\text{MWIS}}$.

Therefore, $\forall n_k$ s.t. $(n_j, n_k) \in E_C \cup E_{I\to C}, n_k \notin N'_{\overline{\text{MWIS}}}$. $N''_{\text{MWIS}} = N'_{\text{MWIS}} + \{n_j\}$ is also an independent set. As $w : N \to \mathbb{R}_{\geq 0}$, $w(N''_{\text{MWIS}}) \geq w(N'_{\text{MWIS}})$, N'_{MWIS} is not guaranteed to be the optimal solution. Contradiction. Therefore, $\forall n_i \in N'_{\overline{\text{MWIS}}}$, if $\langle n_i, n_j \rangle \in E_I$, we must have $n_j \in N'_{\overline{\text{MWIS}}}$.

c) According to a) and b) and the definition of $\overline{\text{AD}}$, $N'_{\overline{\text{MWIS}}}$ is also a solution of $\overline{\text{AD}}$ in $G_{\overline{\text{AD}}}$.

3) According to 1) and 2), we can have $N'_{\overline{\text{AD}}} = N'_{\overline{\text{MWIS}}}$.

□

C. Comparison with D-Disasm and Spedi

D-Disasm [15] is a disassembly framework which implements static analysis and heuristics in Datalog [51]. It features the capability of providing reassembleable assembly. As D-Disasm is originally designed for x86/x64 binaries and only supports 64-bit ARM instructions (A64), we only compare it with D-Disasm on the SPEC-AArch64 dataset. Table X shows the results of comparing D-ARM to D-Disasm on disassembling the AArch64 binaries. The results demonstrate that our approach achieves nearly 100% F1 scores at both the instruction and block granularities. As mentioned in Section II, although 64-bit ARM binaries do not use Thumb instructions, they still have many inlined data, which are difficult for disassembly. D-Disasm generates a lot of false instructions due to failure in recognizing inlined data.

TABLE XI: Comparing D-ARM with Spedi in Precision (P), Recall (R), and F1 scores (%) over Thumb-only binaries

Instructions							
Dataset		**Spedi**			**D-ARM**		
		P	R	F1	P	R	F1
Basic	SPEC2000	99.59	96.60	97.97	99.99	99.94	99.97
	SPEC2006	99.49	95.69	97.41	99.86	99.66	99.76
Data	SPEC2000	85.71	95.52	90.18	98.59	98.34	98.45
	SPEC2006	85.42	94.93	89.28	98.04	98.28	98.13
$r = 0$	SPEC2000	85.17	39.49	52.23	99.99	98.64	99.31
	SPEC2006	85.58	43.89	56.93	99.96	98.11	99.01
$r = 50$	SPEC2000	77.25	15.39	24.49	99.96	98.34	99.14
	SPEC2006	77.55	21.41	32.32	99.94	97.78	98.82
$r = 100$	SPEC2000	76.74	12.85	21.09	99.94	98.08	99.00
	SPEC2006	67.20	11.49	19.13	99.93	97.55	98.69
Reachable Blocks							
Dataset		**Spedi**			**D-ARM**		
		P	R	F1	P	R	F1
Basic	SPEC2000	94.83	82.69	87.64	98.91	99.69	99.29
	SPEC2006	94.46	84.67	88.91	96.62	98.77	97.65
Data	SPEC2000	89.21	79.35	83.36	88.91	96.37	92.32
	SPEC2006	89.27	82.97	85.59	85.70	95.88	89.89
$r = 0$	SPEC2000	26.20	44.36	32.13	94.90	96.08	95.48
	SPEC2006	29.03	49.18	35.26	94.76	95.76	95.25
$r = 50$	SPEC2000	32.89	26.74	27.05	93.30	94.50	93.88
	SPEC2006	30.20	32.48	29.67	93.77	94.99	94.35
$r = 100$	SPEC2000	38.93	27.67	30.88	91.65	92.60	92.10
	SPEC2006	30.38	19.77	22.36	93.22	94.38	93.78

Spedi [35] is a speculative disassembler specifically designed for ARM binaries. It first speculatively recovers all possible basic blocks and then refines them using a conflict analysis. However, Spedi only targets the variable-sized Thumb instructions (16-bit and 32-bit T32, as mentioned in Section II-A). It does not support ARM instructions, let alone the interleaving of mixed instruction sets. For binaries compiled with ARM instructions, Spedi simply decodes the code section as Thumb instructions. Thus, we only evaluate and compare with Spedi on binaries compiled with Thumb instructions, i.e., the half of SPEC-Basic, SPEC-Data, and the binaries built with obfuscation. The results are shown in Table XI. We can observe that, even for the binaries with only Thumb instructions, D-ARM still outperforms Spedi. Although Spedi has good results on SPEC-Basic, its performance degrades a lot on SPEC-Data and the obfuscated binaries. This is because the conflict analysis used by Spedi does not take the inlined data into consideration and is less accurate than our method of maximizing the semantic relations. The results of reachable blocks also show that Spedi misses many code blocks. Spedi also has issues with some large binaries and fails to disassemble them. The failure rates are shown in the supplementary material [18].